SYSTEM BANK

PC98#2

アスキー・システム・バンク(PC98#2)

PC-9801 マシン語入門

岩瀬正幸・藤井敬雄共著



アスキー出版局

まえがき

NECのPC-9801シリーズは，発売以来多大な人気を博し，多くのユーザーに使用されてきました．その種類も，PC-9801，E，F，Mと次々と増え，様々なニーズに対応するようになっており，さらにN88-BASICを中心としたプログラミング環境は，8ビット機だったPC-8801に比べるとかなり速いものになっています．

しかし，周辺デバイスの直接の操作や，大量なデータの高速処理などの機能の実現には，やはりマシン語の利用が不可欠です．

しかしながら，16ビットCPUのマシン語プログラミングは，BASICとはもちろん，8ビットCPUのマシン語と比較しても，かなり難しくなっています．そのためか，従来のPC-9801のマシン語入門書は，CPUである8086のコマンド・リファレンス(マニュアル)で終わってしまった傾向があり，PC-9801で実用になるマシン語プログラムを作成するとなると，物足りない点が感じられました．

本書は，このような観点から，MS-DOSに含まれている様々なマシン語開発ツールを使用して，本格的なマシン語プログラミングを行う方法の解説を目的に執筆を行いました．

まず，CPUの動作と各マシン語命令の解説には，トレース機能のあるデバッガを用いたうえで，意味のある具体的なプログラム例を用いて行うようにすることで，なるべくコマンド説明のみに終らないよう配慮しました．

また，実用的なマシン語プログラムの作成には不可欠なアセンブラの使用法を紹介し，さらにマシン語活用のひとつとして，N88-BASICとのリンク・テクニックと内部ルーチンの利用方法について解説しました．

このような意図が本書においてどれだけ実現できたかは，読者の忌憚のないご意見を待たねばなりません．

紙面の関係で，言葉のつたない点や，初心者の方にはわかりにくい表現があるやもしれませんが，本書によって少しでもマシン語の理解を深めていただければ幸いです．

なお，執筆については，1，2，7，8章とAPPENDIXを藤井が，3，4，5，6章を岩瀬が担当しました．

また，本書の執筆にあたり，ご助言，ご助力くださった関係者各位に紙面を借りてお礼申し上げます．

1985年2月　　　　著者代表　岩瀬　正幸

この本を読むまえに

本書は，NECのパーソナル・コンピュータ，PC-9801シリーズをベースにして，16ビット・マイクロコンピュータ8086のマシン語を学習しようと考えている方のために刊行したものです．

マイクロコンピュータも16ビットともなると，8ビットに比べ，その構造も複雑になり，命令語の数も格段に多くなっています．そのため，マシン語によるプログラミングには，開発用のツールが不可欠になってきます．

本書では，このためのツールとして，デバッガ(DEBUG.COM)とマクロアセンブラ(MASM.EXE)を利用して解説を行いました．

これらのツールは，NECよりPC-9801用に販売されている，MS-DOS ver2.0のディスク中に含まれています．なお，MASMの起動には，最低256Kバイトのメモリが必要ですので注意してください．

本書で使用したおもなハードウェア，ソフトウェアは次の通りです．

○ハードウェア
PC-9801 E/F/M
PC-9881 K
PC-8822

○ソフトウェア
MS-DOS ver2.0
MASM.EXE (Microsoft MACRO Assembler ver 1.20)
LINK.EXE (Microsoft Object Linker ver 2.2)
EXE2BIN.EXE
DEBUG.COM
N88-BASIC(86) ver2.0

また，N88-BASIC(86)ver3.0およびPC-9801U/VF/VMでも，本書のプログラムの動作を確認しています．ご使用の際には，以下の点に注意してください．

○N88-BASIC(86)ver3.0(PC-9801U/VF/VMに付属，PC-9801E/F/Mでは別売)

・文節変換や拡張機能(拡張モニタ，拡張ハードコピーなど)を組み込まない．

・トランスファー・プログラム(リストA-1-1)は，リストのコメントに従ってマシン語のロードアドレスを変更する．

○PC-9801U/VF/VM

・N88-BASIC(86)ver3.0を使用する場合は，上記の変更をする．

・RAMディスク(リスト8-3-1)は，動作しない．

本書の主な内容を紹介します．

1章　マシン語の基礎知識

マシン語に初めて触れる人のために，マシン語の実体と構造，マシン語プログラムとBASICプログラムの速度比較，マシン語プログラム開発の手順などをまとめました．また，本書で使用するツール・ソフトウェアについて紹介しました．

2章　CPU（8086）の基礎知識

16ビットCPUである8086の特徴と内部構成，命令の種類とモードについて解説しました．とくに，8086に特有のセグメントの概念については詳解してあります．

3章　CPU（8086）の基本動作

8086の基本的な内部動作について，MS-DOS上のマシン語開発用ソフトウェアであるデバッガを使用して具体的に解説しました．また，デバッガとPC-9801のN88-DISK BASIC（86）に付属されている，機械語モニタとの相違点についても触れました．

4章　マシン語プログラミング

8086の各命令の用法と意味を，目的をもったプログラムを作成することを通じて，具体的に解説しました．また，プログラムを作成する際に基本となるアルゴリズムの表記法について紹介しました．

5章　アセンブラによるマシン語の開発

本格的なマシン語プログラムの作成には不可欠なアセンブラについて，MASMを例にその概要と用法を解説し，第4章で使用したマシン語プログラムをアセンブリ表記したものを，実行可能なマシン語コードにするまでの手順を紹介します．

6章　BASICとマシン語プログラムのリンク

マシン語と高級言語のリンクの例として，N88-BASIC（86）で記述されたプログラム中からマシン語ルーチンをコールする各種の方法について解説します．また，BASICインタープリタ内のルーチンをユーザーが作成したマシン語プログラムから利用する方法について解説します．

7章　ハードウェア構成と割込み

PC-9801のハードウェア構成の概略とI/Oポートについて紹介したうえ，ハードウェア割込みの手順と利用方法について解説します．また，ハードウェア割込みの例として，プリンタスプーラ・プログラムを紹介します．

8章　マシン語ランダム・テクニック

この章では，マシン語学習のまとめとして，いくつかのプログラムの紹介を通して，実用となるプログラムを作成する際のテクニックを解説します．

入力に関するテクニック――――キー入力ルーチン
演算に関するテクニック――――行列演算ルーチン
ディスクに関するテクニック――RAMディスク
言語に関するテクニック――――簡易インタープリタ

APPENDIX 1 MS-DOS・BASICトランスファー・プログラム

MS-DOSディスク上のマシン語プログラムを，BASICディスク上に転送するプログラムを紹介します．本書に掲載されたプログラムで，BASICから使用されるものは，このプログラムで転送します．

APPENDIX 2 MASMの高度な利用法

本書で紹介したMicrosoft MACRO Assemblerの，本文中で解説しきれなかった高度な利用法について解説を加えます．

また折込み付録として，8086の命令一覧を付けました．

なお，本書に掲載されたプログラムを,「ディスクアルバム4」として発売しています．
入力の手間とミスを省き，マシン語プログラムの理解を深めるための助力としてご利用ください．

目　次

1章
マシン語の基礎知識

どの言語でもそうですが，マシン語にも長所と短所があります．マシン語を勉強しようとしている方は，きっと長所を知っていて，それを生かしたプログラムを作りたいと思っているはずです．

この章では，マシン語の構造と，マシン語プログラム作成の手順，及びそれに必要なツールについて説明することにより，後の章で進めるマシン語の勉強がスムースにできるようになることを意図しています．

1-1　マシン語とは

1-1-1　マシン語の実体

よく「マシン語は難しい」といわれます．マシン語もBASICなどと同じコンピュータ用言語の1つなのに，なぜこれだけが難しいといわれるのでしょうか．また一方では，マシン語は単純であるともいわれ，さらに，マシン語は便利であるといわれることも事実です．

このようにいろいろな評価をされるマシン語とはいったい何か，それを見極めることから話を進めていきましょう．

その手はじめとして，マシン語の実物を見てみることにします．PC-9801に内蔵されているROMと呼ばれるメモリには，マシン語で書かれたBASICインタープリタ・プログラムが書き込まれていますので，それを機械語モニタのDコマンドで表示してみました（図1-1-1）．

```
0000 FA B4 D5 9E 79 2B 75 29 7B 27 73 25 F8 3F 73 21   ﾄｳ｢y+u)(`s% ?s!
0010 B0 01 B4 02 F6 E4 70 19 33 C0 9E 78 14 74 12 7A   ｰ ｴ ﾄ◢p 3ﾀ｢x t z
0020 10 72 0E F9 3F 72 0A 80 7F 84 20 F6 E4 71 02 EB    r ?r ｰ ｴ ﾄ◢q ♣
0030 01 F4 B8 FF FF 8E D8 8C DB 8E D3 8C D1 8E C1 8C    ｸ ■ﾘ■ﾛ■ﾓ■ﾑ■ﾁ■
0040 C2 8B E2 8B EC 8B F5 8B FE 3B F8 74 01 F4 0B FF   ﾂ■ ■ ■ ■ ; t
0050 74 05 2D 55 55 EB DE BB 4F 08 B4 60 BC 62 00 E9   t -UU♣ﾞｻO ｴ`ｼb ♥
0060 43 06 BB 51 08 B4 20 BC 6D 00 E9 38 06 BB 53 08   C ｻQ ｴ ｼm ♥8 ｻS
0070 E4 33 A8 08 75 03 8B 5D 08 B4 20 BC 81 00 E9 24   ◢3ｨ u ｻ] ｴ ｼ_ ♥$
0080 06 B4 09 E4 33 A8 08 75 02 B4 08 8A C4 E6 6B B0    ｴ ◢3ｨ u ｴ ■ﾄ◥kｰ
0090 44 E6 11 B0 92 E6 37 B0 07 E6 37 E6 50 B8 00 F0   D◥ ｰ ◥7ｰ ◥7◥Pｸ ×
00A0 8E C0 33 C0 BB 00 80 B9 00 40 26 02 07 43 26 02   ■ﾀ3ﾀｻ _ｹ @&  C&
```

図1-1-1　マシン語の表示

これは，1バイトの内容を2桁の16進数で表示したもので，我々が見ることのできるマシン語プログラムの実体です．人間の使う言葉と大きく異なって，数字の集まりといった印象があるでしょう．

ところで，実はここに示されたものもマシン語本来の姿ではないのです．CPUが解釈，実行できるのは，電気信号によるパルス（マシン語コード）だけです．このコードを2進数の1と0に置き換え，この1と0の組合せを8つ（8ビット）まとめてひと区切り（1バイト）とし，さらにそれを2桁の16進数で表現したものがこの出力例なのです．

このことからも，マシン語がいかに機械向きの言語なのかが理解されると思います．どうもこの人間になじみにくい形態が，我々にマシン語を難しく感じさせる一因になっているようです．

1-1-2　マシン語とBASIC

パーソナル・コンピュータに始めて触れる人は，ほとんどがBASIC言語を使用してプログラムを作成することだろうと思います．そしてBASICをある程度マスターするか，あるいは，BASICの限界に突き当たる頃，マシン語に興味をいだきはじめるのが通例のようです．

そこで，この項ではマシン語とBASICを比較し，その相違点を知ることを通して，マシン語とマシン語プログラムの特徴を知ろうと思います．

次の2つのリストを見てください（リスト1-1-1，1-1-2）．

リスト 1-1-1　BASICプログラム1

```
100 FOR I=1 TO 60000!
110 NEXT
```

リスト 1-1-2　マシン語プログラム1

```
0000 B960EA
0003 E2FE
```

これは，60000回のループをするプログラムをBASICとマシン語で表記したものです．PC-9801上でこれらのプログラムを実行すると，BASICプログラムが30秒もかかるのに対し，マシン語プログラムではわずかに0.16秒しか，かかりませんでした（クロック8MHzの場合）．

この差はどこからくるのでしょうか？

マシン語はCPUによって直接実行される言語です．この実行の様子を図1-1-2に示します．この図は，同時にCPUの動作を表していると考えることもできます．すなわちCPUはメモリ中から命令を読み込み，それを解釈して実行するということの繰返しを行っているのです．1つの命令はマイクロ秒（10^{-6}秒）オーダーの，ものすごいスピードで解釈，実行されていきます．マシン

語プログラムの実行が速いのはここに理由があるのです．

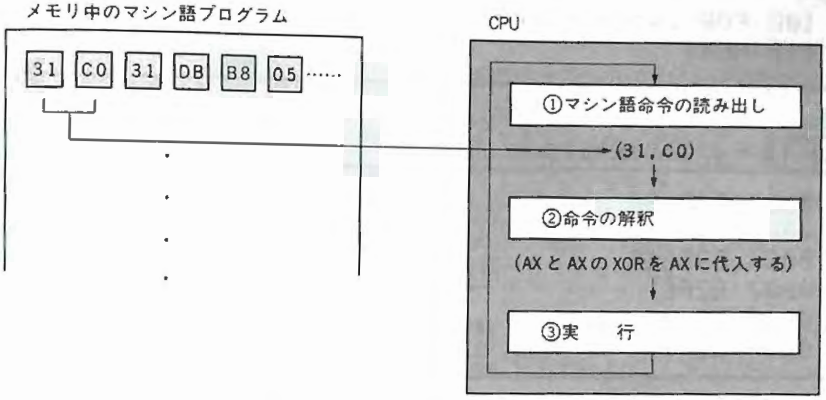


図1-1-2　マシン語の実行のされ方

章頭で述べたように，PC-9801 の BASIC インタープリタもマシン語で書かれています．インタープリタとは，逐次翻訳方式の処理を行うプログラムのことで，BASIC インタープリタは BASIC 言語の文法で書かれたプログラムを，各命令ごとに解釈して実行していきます．この様子を図1-1-3 に示します．

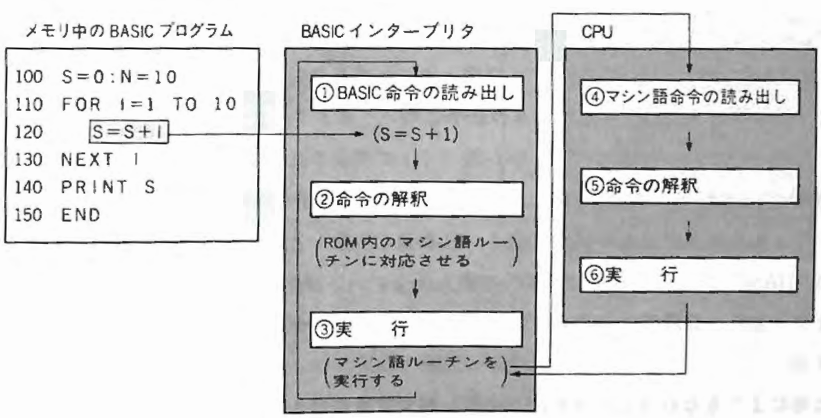


図1-1-3　BASIC プログラムの実行のされ方

前述のように，マシン語は CPU によって直接実行されますが，BASIC 言語は，マシン語で書かれたプログラムによりいったん解釈されてから実行されるのです．このように，インタープリタでは，各命令を逐次解釈しなければならないため，実行の速度が格段に遅くなってしまうのです．

しかし，一方で，BASIC にはそれなりの利点があります．たとえば，さきほどのプログラムのループ回数を 70000 回にしてみましょう（リスト 1-1-3，1-1-4）．

リスト 1-1-3 BASICプログラム 2

```
100 FOR I=1 TO 70000!
110 NEXT
```

リスト 1-1-4 マシン語プログラム 2

```
0000 B90A00
0003 51
0004 B9581B
0007 E2FE
0009 59
000A E2F7
```

どうですか？ BASICプログラムがわずかの書変えで済むのに対し，マシン語プログラムは大幅に書き変っています．それだけプログラム上の制約が多いことを示しているのです．

また，BASICプログラムはインタープリタにより管理されているので，プログラムの実行途中で自由に停止させることも可能ですが，CPUが，ただ順番に命令を実行するマシン語プログラムではそうはいきません．プログラムのミスにより，実行の制御が不能になることを暴走といいますが，マシン語プログラムがよく暴走を起こすといわれるのはそこからくるのです．つまり，「ちょっと試しに」という気楽な気持では通じないというわけです．マシン語が難しいと思われるもう1つの原因が，このようなBASICでの環境と比べた場合の対話性のなさにあるといえるでしょう．

次にマシン語とBASIC言語の個々の命令について考えてみます．

マシン語の1つの命令は1バイトから数バイトで構成されます．しかしその機能は，メモリやCPU間のデータ転送，簡単な演算，実行順序の変更（分岐）といった程度のものがほとんどなのです．つまり，個々の命令の能力は低く，仕事量が少ないという欠点があります．

一方，BASICの1つの命令は，実行の際大きなマシン語ルーチンに展開されて，強力な機能を持ちます．また，普段我々が用いている言葉に近い表現を使うことができるという長所もあるわけです．

これらによりもたらされる利点は，マシン語でプログラムを組む場合に比べて，より抽象的にプログラムを考えることができるということです．たとえば関数電卓のもつ機能をプログラムする時，BASICであれば三角関数や対数関数などの基本的な関数がそのまま使えますから，ほとんどメインルーチンを書くだけで完成しますが，マシン語で最初から作ろうとすれば，関数を実現するサブルーチンはもちろん，浮動小数点の表現など多くの問題を解決しなければなりません．つまりマシン語の場合には，命令が貧弱であるため，メインとなるアルゴリズム（問題解決の方法）以外にいろいろと気をつかわなければならないのです．

しかしこの反面，BASICに比べハードウェア寄りの細かい操作にはマシン語のほうが向いています．

以上の点をまとめると次の結論が出せます．

○マシン語は実行速度が速く，CPUが直接実行するため細かい動作が可能である
○しかし，個々の命令の能力が低いため，役立つ作業をするプログラムを作るには多量のステップを必要とするうえ，操作環境が悪く，楽なプログラミングができない
○したがって，マシン語プログラムの開発には，有効なツールを用いるべきであり，また速度を重視しない部分は，BASICなど操作性のよい言語と組み合せる方が得策である

本書もこの考え方にそって解説することにしましょう．

1-1-3　マシン語プログラムの構造

マシン語の各命令文は1から数バイトで構成されていますが，その内容は命令の種類を表す部分，いろいろな属性を表す部分とデータ部分から成っています．

マシン語プログラムは，これらの命令文が図1-1-4のように並んだものです．この図ではメモリの0番地からプログラムが始まっていることにします．

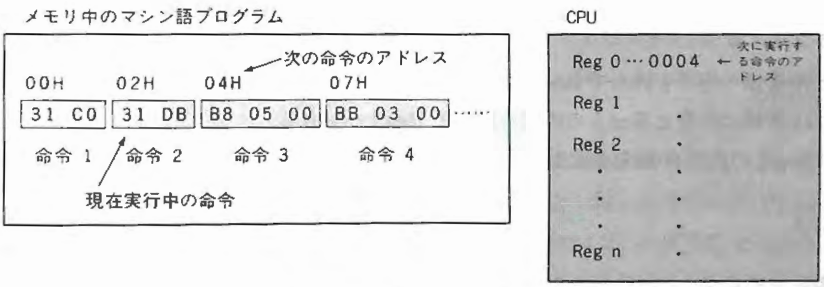


図1-1-4　メモリとレジスタ

さて，BASICでは，行番号というもので実行の順序が制御されていましたが，マシン語ではそのような概念はなく，実行順序はメモリのアドレス（番地）で考えられています．したがって，CPUが次々と命令を実行していくためには，その命令のあるアドレスをCPUが知っていなければならないのです．このためCPUの内部には，メモリと同じような記憶回路があり，実行される命令の入ったアドレスを保持しています．

CPUにはこの他にも，処理されるデータの入ったアドレスを示したり，データの一時的な保持や演算を行うための記憶回路があります．これらの記憶回路はレジスタと呼ばれ，マシン語プログラムでちょうどBASICの変数のような働きをします．図1-1-4では，Reg 0, Reg 1,……, Reg nでこれらを示してあります．

レジスタの数や構成はCPUによって異なり，CPUを特徴づける要素にもなっています．レジスタには，我々がマシン語命令中で汎用的に扱うことができるものと，そうでないものがあります．たとえば，次に実行する命令アドレスを保持しているレジスタ(図ではReg 0)は，通常プログラムの流れに従い，CPU内で自動的に更新されていきます．我々がこのレジスタの内容を変えることは実行順序の変更，すなわち分岐を行うことになりますから，データの保存などのためにこのレジスタを使うことはできないのです．

マシン語プログラムでは，高級言語のように，任意の変数名を付けた変数を扱うことはできないので，演算などで使用する数値データ等は，その格納されているメモリのアドレスをもとに参照することになります．そのためにマシン語の命令は，あるアドレスのデータを，いろいろな指定の方法でアクセスできるようになっています．たとえばアドレスの指定に直接その値を示したり，あるアドレスからの相対的な位置で示したりすることができるのです．

このアドレス指定のことをアドレッシングと言い，マシン語プログラムでは常に考えなければならないことの1つです．

ここで，マシン語による演算について考えてみましょう．マシン語プログラムでの演算はレジスタ同士で行うか，レジスタとメモリ間で行うかのどちらかです．

例として10番地と11番地の1バイトデータを加えて12番地に格納する処理を2つの異なったアルゴリズムで表してみましょう．

アルゴリズム 1(ダイレクト・アドレッシングによる)

10番地の内容を読んでReg 1へいれる
11番地の内容とReg 1の内容を加えてReg 1へいれる
Reg 1の内容を12番地にいれる

または，

アルゴリズム 2(インダイレクト・アドレッシングによる)

Reg 2に10というアドレスをいれる
Reg 1にReg 2の内容 (10) の示すアドレスの内容をいれる
Reg 1とReg 2の内容+1 (11) の示すアドレスの内容を加えてReg 1にいれる
Reg 1の内容をReg 2の内容+2 (12) の示すアドレスにいれる

アルゴリズム1では，絶対的にアドレスを指定していますから，このアドレスに対してしか処理を行うことができません．ところがアルゴリズム2では，Reg 2にいれておく値を変えるだけで，同じような処理を異なるアドレスに対して行うことができるのです．

以上のことからも，マシン語プログラムでは，メモリとレジスタの使い方と，命令の種類，アドレッシングということがらが，プログラムを構成するうえで重要になっており，プログラムの構造もこれらによって決定されることが理解できるでしょう．

1-2　マシン語プログラム開発の手順

1-2-1　マシン語とアセンブリ言語

前節で説明したとおり，16進数によるマシン語の形態は，人間にとって扱いにくいものでした．そこで我々がプログラムを考える際には，各命令に対してその命令の動作を表す人間になじみやすい表記を対応させてやることが考えられます．

リスト1-2-1にその例を示しますが，左のマシン語を見てその意味を思いうかべるより，右のシンボルで考えるほうがずっとわかりやすいでしょう．これらのシンボルはニーモニックと呼ばれ，命令の種類を表すオペコード部分と動作の対象などを示すオペランド部分から成り立っています．マシン語の各命令と1対1に対応したニーモニックで表現する言語をアセンブリ言語と言い，我々がマシン語プログラムを作る場合は，このアセンブリ言語を使って考えていく方が能率が上がります．

リスト 1-2-1　マシン語とニーモニック

```
マシン語コード      ニーモニック
BF0000             MOV      DI,0000
B0FF               MOV      AL,FF
B90001             MOV      CX,0100
F3                 REPZ
AA                 STOSB
```

この本のマシン語命令の解説は，以降，このニーモニック表現を使用することにします．

ところで，アセンブリ言語によるプログラムは，マシン語と1対1に対応していても，言語自体はマシン語と違い，ただの文字列の集まりです．そこで実際にCPUに実行させるためには，各命令をマシン語のコードに変換することが必要になります．この操作はアセンブルと呼ばれ，人間が命令表（折込み付録参照）を見て行うハンド・アセンブルと，次に紹介するアセンブラというプログラムで行うものの2とおりの方法があります．

1-2-2　マシン語開発に必要なツール

BASICのプログラムは，行番号に続いて命令を入力していき，それが終ったら"論よりRUN"として，デバッグしながら完成に近づけていくことができます．これはこのBASIC処理系にプログラムを入力，編集する機能と，デバッグのための機能が，翻訳処理機能とともに用意されているため実現されるのです．

マシン語にはこのような機能はありませんから，プログラム開発にはいくつかの道具を使用することになります．

○ローダ

マシン語プログラムを実行する場合に最低限必要なのは，マシン語プログラムをメモリ内にセットすることです．この操作を行うものをローダ（LOADER）と言い，メモリにハードウェア的に電気信号を直接書き込んだり，紙テープやディスクから読んでメモリに書いたり，キーボードからハンド・アセンブルしたマシン語を入力するといったことを行います．このローダはすでにできあがっているプログラムに対して使うもので，プログラム作成を助けるものではありません．

○アセンブラ

プログラムを効率よく作成するためには，アセンブリ言語によるプログラムを機械的にマシン語に変換してくれる，アセンブラという変換プログラムを使用するとよいでしょう．なぜならマシン語では，アドレスというものを常に意識しなければならないからです．たとえばリスト1-2-2(a)のプログラムで，誤りが見つかってリスト1-2-2(b)のように命令が1つ増えたとしますと，それ以降の命令アドレスがずれてしまいます．つまり命令をずらしてもう一度セットしなおさなければならなくなるのです．

リスト 1-2-2(a)

```
0000 B8FF00        MOV     AX,00FF
0003 4A            DEC     DX
0004 75FD          JNZ     0003 ←――― ジャンプ命令
0006 48            DEC     AX
0007 75FA          JNZ     0003 ←――― ジャンプ命令
0009 C3            RET
```

リスト 1-2-2(b)

```
0000 B8FF00        MOV     AX,00FF
0003 BAFF00        MOV     DX,00FF ←―― 挿入した命令
0006 4A            DEC     DX
0007 75FD          JNZ     0006
0009 48            DEC     AX
000A 75F7          JNZ     0003
000C C3            RET
```

これではキーボードなどから入力していた場合に大変な手間になります．しかも悪いことに，同じ命令をずらすだけではすまず，命令を修正する必要も出てきます．リスト1-2-2では，追加された命令の後に実行を移すジャンプ命令（BASICのGOTO文に対応）がありますので，その飛び先を示すアドレス部分を書き換えなければなりません．アセンブラでは，ただ1対1に命令をおきかえるだけでなく，このような場合の変更（アドレス計算）も行ってくれます．

また，マシン語命令の機能の低さを擬似的に補うマクロ機能や，構造化機能を持つアセンブラもあり，マシン語を直接扱うよりも格段の能率の向上が得られます．

○エディタ

アセンブラを使うには，アセンブリ言語で書かれたプログラム（ソース・プログラムと呼ぶ）を用意しなければなりません．これは，単なる文字の集まりですから，このソース・プログラムの作成には文字を編集するツールであるエディタ（EDITOR）プログラムを用います．このエディタは，行単位での編集向きのラインエディタや，BASICでおなじみのスクリーンエディタなどのいくつかの種類に分類できます．

○リンカ

長いプログラムを作成する場合，サブルーチンごとにいくつかの部分に分けてプログラミングしたり，入出力ルーチンなど汎用性のある部分は別個に作成しておいて，後でつなげたりしたい場合が応々にしてあります．このようなときに使用されるツールが，リンカ（LINKER）と呼ばれるプログラムです．リンカでつなげられる汎用ルーチンのことをライブラリ（LIBRARY）などと呼びます．リンカを使用すれば，マシン語プログラムにつきものの，アドレス合せという面倒な作業から逃れることができます．

リンカは必ず使用するものではありませんが，用意されていると便利なツールのひとつといえるでしょう．

○デバッガ

以上で説明したエディタ，アセンブラ，リンカ，ローダで，アセンブリ言語によるプログラムの作成からマシン語プログラムの実行までができます．残るは虫とり（デバッグ）のためのツールですが，これにはデバッガと呼ばれるプログラムを使います．デバッガは高級なモニタというようなもので，マシン語プログラムのロード，セーブ，実行などの機能のほかに，ブレーク・ポイントの設定やトレース，簡単なアセンブル，逆アセンブル（マシン語からニーモニックを生成する）などの機能があります．

簡単なプログラムであれば，デバッガで十分に作成することができるのと，CPUの動きを目で見て確認できるため，本書では，第4章までこのデバッガを使ってプログラムの説明をしていきます．

1-2-3　マシン語プログラム開発の手順

マシン語プログラムでも，他の高級言語プログラムでも，開発を行う時にまず考えなければならないのは，その問題を解決するための方法，すなわちアルゴリズムです．アルゴリズムの表現には，フローチャートなどの図を用いる方法や，言語仕様のはっきりした高級言語を用いる方法などがあります．

そのうえ，マシン語のプログラムでは，各命令の機能が単純であるために，本来のアルゴリズム以外にいろいろと気を使う必要があります．またマシン語特有のテクニックがいくつか考えられますが，多くの人が使っていてすぐそれとわかるもの以外，たとえばプログラムの実行中に，そのプログラム自身の命令の一部を書き換えるようにして別の仕事をさせる，などという複雑な手法は極力使わないように心がけたほうが良いでしょう．

アルゴリズムをプログラム化することをコーディング(CODING)と言います．マシン語プログラムの作成ではアセンブリ言語でこれを行うわけですが，これは，一度紙上で書きあげてしまってからエディタで入力するのも良いし，慣れれば，エディタで直接入力していっても良いでしょう．この部分は，先のアルゴリズムをきちんと考えておけば，そんなに時間のかかるものではありません．

エディタで入力，編集の終わったソース・プログラムは，アセンブラでアセンブルすることにより，マシン語プログラム（コード）になるわけですが，アセンブラによっては直接実行可能なコードを生成せず，再配置可能（リロケータブル）な形式のコードを出力するものがあります．

リロケータブルなプログラムとは，ロードするアドレスや，プログラム中で参照するアドレスなどが実行時に設定されるプログラムで，実際に実行されるコードに加えて，アドレスを変更する情報が付け加えられたものです．簡単に言えば，後でどのアドレスにもロードできるプログラムのことを指します．

この形式を用いると，マシン語プログラムのライブラリ化が可能になります．つまり，リロケータブルなコードで汎用のサブルーチンなどを作っておけば，それを使いたい時には，リンカなどで指定してメインルーチンとくっつければそれで済むのです．この操作をリンクといいます．

もしリロケータブルなコードが得られない場合には，そのルーチンをアセンブリ言語レベルで編集して，再度全体をアセンブルしなおさなければなりません．

アセンブルされ，リンクの終ったプログラムは，メモリ中にロードして実行することができます．

ここまでの作業でエラーの発生しなかったプログラムも，実行する際には暴走を起こす可能性があるということに注意しなければなりません．アセンブラやリンカでは文法的なミスは指摘してくれますが，アルゴリズムの誤りまでは責任を持ってくれません．

通常，リンクの終ったプログラムが思ったとおりに動かなかった時には，アセンブリ言語レベルでの検討を行うか，デバッガを使ってレジスタの動きなどを追って考えていきます．マシン語プログラムの作成という作業は，人間の頭の中でCPUの動きを考えていくことですから，デバッガを使いこなせれば効率が上がるはずです．

最後に，今まで述べた操作を図1-2-1にまとめておきます．

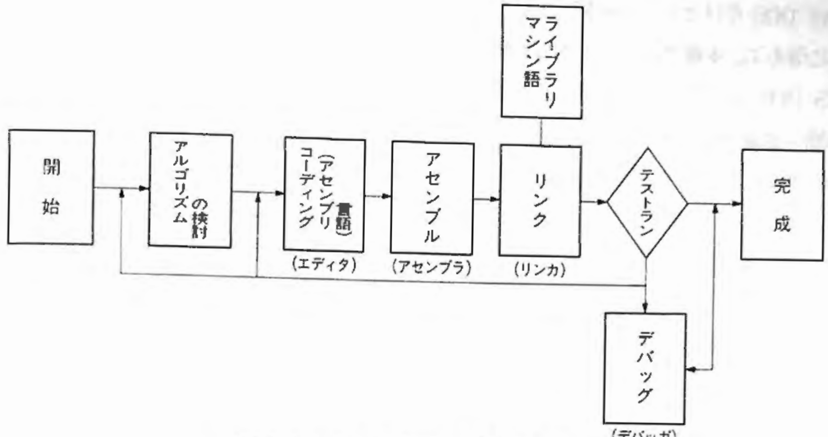


図1-2-1　マシン語プログラム開発の手順

1-3　本書で使用するツール

1-3-1　MS-DOS

前節で述べたように，マシン語の開発には，それに適したツールを使用して，機械にできることは機械にまかせながら流れに従って作成する方が，精神的にも肉体的にも楽です．

ところで，前出のような様々なツール・ソフトウェアを，次々と切り換えて実行していくことをBASIC上で行うことは困難です．

なぜなら，BASIC上で走るそのようなツール・プログラムがほとんどなく，あっても汎用性の点で何らかの難点があるからです（もちろん自作するなどは問題外とします）．

そこで登場するのが，MS-DOSという汎用的な環境を作り出すソフトウェア（OS：オペレーティング・システム）です．このMS-DOSは，米国マイクロソフト（Microsoft）社の開発した，ディスク用のOS（DOS）で，一言で言うなら，異なるハードウェア上でも同じソフトウェアが走るようにハードウェアとソフトウェアの仲立ちをするプログラムのことです（図**1-3-1**）．

PC-9801用のMS-DOSは，NECから1万円程度で発売されていますが，特筆できるのはこの

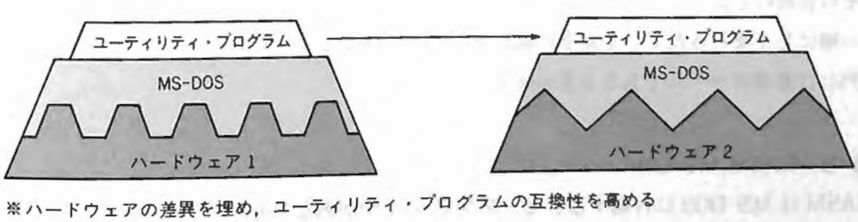


図1-3-1　MS-DOSの働き

中に MS-DOS だけでなく，前述したようなマシン語開発用のツールがすべて含まれているのです．

このことから，本書ではこの MS-DOS を使用して，マシン語の解説を進めていきたいと思います．MS-DOS をお持ちでない方にも理解できるよう努力しますが，サンプル・プログラムなどに具合の悪い点があると思いますし，16 ビットマシンのマシン語開発にはツールが不可欠であることと合わせて，可能なかぎりお求めになることをおすすめします．

ところでこの本で扱うソフトウェアは，図 1-3-2 で示したようになっています．作成するマシン語プログラムは図の灰色で表されている部分で，ハードウェアにかかわる部分と，BASIC に関係するところです．開発は DOS 上で行っても，実行させるためには，BASIC の管理する環境へ移さなければなりません．このためのツールとしては，APPENDIX にファイル転送用プログラムを用意してあります．

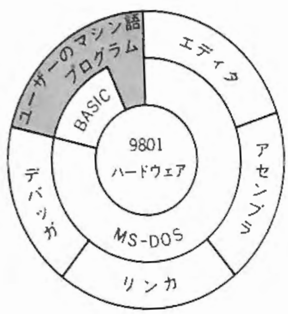


図 1-3-2　マシン語のためのツール・ソフトウェア

MS-DOS は，システムのはいったディスケットをドライブにセットして，電源を入れるか RESET することにより起動します．起動後，コマンド・プロンプトと呼ばれるメッセージが画面に表示されれば，これから紹介するツールをディスクから任意に呼び出して使うことができます．

MS-DOS の働きについて追究するのは本書の目的ではありませんので詳解はしませんが，興味のある方は，「入門 MS-DOS」（アスキー）等を一読してください．

1-3-2　DEBUG

DEBUG（デバッガ）は MS-DOS 上のデバッガ・プログラムです．本書では第 3, 4 章でデバッガを用いて CPU の動作を説明し，簡単なマシン語プログラムを作成します．デバッガは，本質的にはその名前のとおり，プログラムの誤り（バグ）を発見，修正するためのツールですが，実際には一瞬にして実行されていく命令の動作を 1 つずつ確認していくトレース機能があり，マシン語入門には最適のツールであると思います．

1-3-3　MASM, MS-LINK

MASM は MS-DOS に付属するアセンブラで，いくつかあるアセンブラのうちでも高度な機能を持つ部類に属し，それゆえ初心者には扱いづらいと言われていますが，基本的な用法さえ理解

すれば大変有効なツールとなります．

MASM の出力するコードは，MS-LINK (LINK . EXE) というリンカによって MS-DOS 上で実行可能なプログラムになります．

MASM を用いたマシン語プログラム開発は，第5章から説明します．本書では紙数の都合もあり，これらのツールの能力を 100 %いかすような使い方までは言及していません．ある程度理解ができた後，徐々に高度な機能を利用していくと良いでしょう．

1-3-4　EDLIN

EDLIN（エドリン）は，文字列を編集するエディタ・プログラムです．アセンブリ言語のソース・プログラムだけでなく，普通の文章や他の言語のソースも作成，修正することができます．

EDLIN はラインエディタという形式のもので，行の挿入，削除，着目行の移動，表示といった方法を最低限マスターすれば，いちおうの編集作業が行えます．加えて，編集するファイルの読込みと，終った後の書込みの方法を知っておけば，目的のプログラムを作成することができます．

EDLIN の使用方法も第5章で解説します．

1-3-5　BASIC モニタ

BASIC モニタは DISK-BASIC 上で MON コマンドを実行すると起動するマシン語入力用のユーティリティです．機能的には，MS-DOS の DEBUG に似ていますが，ミニアセンブラ，逆アセンブラがすべての命令をサポートしていないことや，MASM と異なるニーモニックになることなどから，本書ではこのモニタを開発ツールの中心としては扱っていません．しかし，MS-DOS 上で開発したマシン語を，最終的に BASIC で用いる際のデバッグには有効です．

モニタ・アセンブラの使用方法は，「PC-9801 ユーザーズ・マニュアル」の第17章に記載されていますので，本書では，デバッガとの表現の違いについてのみ解説するにとどめます（3-3 参照）．

2章
CPU(8086)の基礎知識

この章では，PC-9801に使用されているCPU（8086）の特徴と概略について解説します．

第1章で述べたように，マシン語プログラムでは，命令の機能はもちろんアドレスの指定方法やレジスタの構成，その使い方などが重要な要素になります．そこで本章の前半では8086の一般的な特徴とアドレス空間について述べ，後半でレジスタ構成や命令の種類について説明していきます．

2-1　8086の特徴

2-1-1　8080から8086への変遷

マイクロコンピュータの本格的な利用は，8ビット汎用CPUの出現により急激に広まりました．

1973年にインテル（Intel）社が他社に先がけて開発を開始した8ビット汎用CPUの8080は，現在よく使われているZ80の原型ともいえるCPUで，いわゆる80系という大きな流れの基を形作るものになりました．マイクロソフト社のBASICやデジタルリサーチ(DIGITAL RESEARCH)社のCP/Mなどのソフトウェアも，この8080 CPU上で最初に動きだしたのです．

8080の改良型である8085やザイログ(Zilog)社のZ80は，ソフトウェア的に8080の上位互換性を持っているため，ソフトウェアの資産をそのまま引きつぐことができました．

CPUの性能を向上させるには，命令の機能を強化する方法と，命令の実行速度を上げるということがまず考えられます．たとえばZ80は，8080の命令に加えて強力な命令が追加され，最近では8MHzのクロック（動作周期）で動作するものも供給されています．

このような方法と平行して考えられる性能向上の策としては，処理単位を大きくする，すなわち内部の演算単位を16ビットや32ビットにするという方法があります．8086は，インテルが生み出した最初の16ビットCPUのひとつで，8085やZ80のような8080とのソフトの互換性はありませんが，命令の体系が似ているため，80系のマシン語を使用していたプログラマーは割合楽に8086を使い始めることができます．

8086が，8080などの8ビットCPUと比べて強化された主な点をまとめると次のようになります．

○メモリ空間が拡張された

8086には，アドレスを指定するための信号線が20本出ているため，1Mバイト($2^{20}=1048576$)までのメモリを直接アクセスすることができます．1Mバイトのメモリ空間は，64Kバイトを単位とするセグメント値とその中でのオフセット値を与えることにより，指定するようになっています．

○強力な命令が追加された

まず16ビット単位で処理を行う命令が増えたことがあげられ，さらに，乗除算命令や，ストリング命令と呼ばれるブロック転送，データ検索などに便利な命令が追加されました．

○処理速度が向上した

半導体技術の進歩により，16ビットCPUでは5, 8, 10MHzなどのクロック周波数で動作するものが作られています．また同じクロック周波数を与えた場合でも，内部の処理の違いにより8ビットCPUよりも高速の処理を行います．

○アドレッシング能力の強化

広いメモリ空間を有効にアクセスしたり，いろいろなデータ構造を処理するプログラムが効率よく実現できるように，アドレッシングモード（命令の作用するアドレスの指定方法）が多く用意されています．

8086には，以上の他に，8080で使われた周辺デバイス(LSI)を利用できることや，CPUのマルチ（複数）化が考慮されている点に特徴があります．

2-1-2 メモリ空間とセグメント

8086の1Mバイトのメモリ空間を図で示すと，図2-1-1に示すように00000H～FFFFFHの5桁の16進数で表される連続した領域になります．プログラムやデータはすべてこの領域内に置かれるわけです．

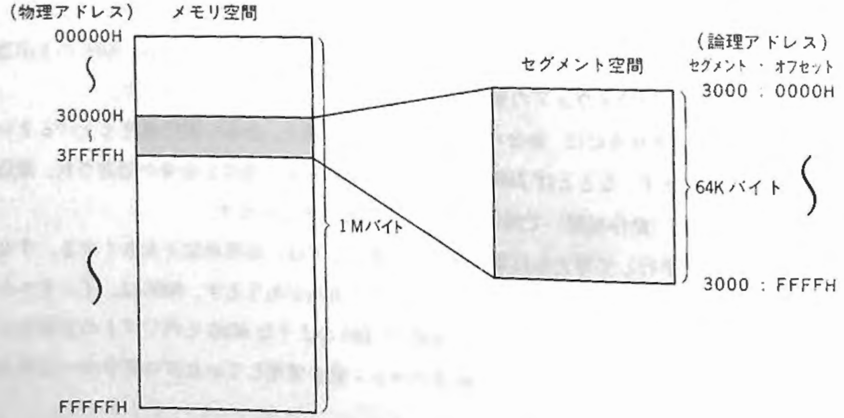


図2-1-1 メモリ空間とセグメント

ところで，CPUが命令を実行するためには，その命令のあるアドレスを内部に保持しておく必要があることは第1章で述べました．ところが，8086内の記憶回路（レジスタ）はすべて16ビット（64Kバイト分）しかなく，1つのレジスタでは20ビットの値を表現することができないのです．

この問題を解決するために，2つのレジスタを組み合せて20ビットの値を得る方式が考え出されました．つまり1Mバイトの領域を16バイト置きに示すレジスタ（セグメント・レジスタ）と，そのレジスタをベースとしてそこからの64Kバイト分を示すレジスタ（オフセット・レジスタ）とを用意して，この2つで20ビットのアドレスを示そうというわけです．

セグメント・レジスタを利用したこのようなアドレス表現を，論理アドレス，16進5桁で表現した絶対的なアドレス表現を物理アドレスと呼び，我々がマシン語プログラムを作成する際は，すべて論理アドレスを用いて記述しなければなりません．

論理アドレスから物理アドレスを求めるには，セグメント・アドレスを16倍（16進で1桁繰り上げる）して，オフセット・アドレスを加えることにより行います．例を図2-1-2に挙げました．

逆に物理アドレスからは図2-1-3のように無数の論理アドレスの組合せを求めることができますが，これらはわかりやすいものを適宜選んで使用することになります．本書ではプログラムの先頭のオフセット・アドレスが0000Hか0100Hになるようにセグメントを決めるようにしました．

```
  セグメント・アドレス：1234H
  オフセット・アドレス：5678H
のとき
  物理アドレスは

       12340H
   +)   5678H
  ─────────────
       179B8H
```

図2-1-2　物理アドレスの生成

```
物理アドレス：12345Hのとき
①セグメント・アドレス：1000H
  オフセット・アドレス：2345H
②セグメント・アドレス：1234H
  オフセット・アドレス：0005H
③セグメント・アドレス：1111H
  オフセット・アドレス：1235H
      ⋮
（以下，何通りもの表現の仕方がある）
```

図2-1-3　論理アドレスの生成

2-1-3　メモリへの格納のされ方

次に命令やデータの格納状態をみてみましょう．

8086マシン語命令や，扱うデータのメモリ内での格納状態で1つ注意しなければならないことは，16ビット（2バイト）で取り扱うデータがメモリ中におかれる場合，下位バイト，上位バイトの順に格納されることです．たとえば図2-1-4の1000H番地からにある命令は，1234Hをレジスタに格納する命令ですが，メモリ中で，この命令は図のような順番で格納されています．また2000H番地から5678Hという16ビットデータが格納されているとすると，通常，図のように78H，

56H の順になっています．

メモリ中で16ビットで扱うデータが下位，上位の順に格納されるのは，Z80 などの 80 系 CPU すべてに共通することです．アセンブラでマシン語プログラムを記述する際，2バイトのデータを2バイト単位で扱っている時には，このようなメモリ中の状態を意識することはありません．なぜならば，すべての2バイトデータをアクセスする命令が，このような形式でデータを扱ってくれるからです．ただし，2バイトデータを1バイト単位で扱うような場合は注意が必要です．

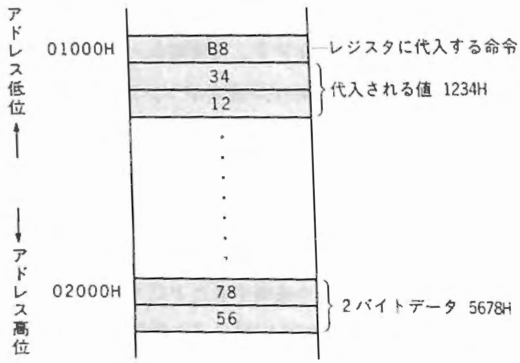


図 2-1-4　メモリへの格納のされ方

2-1-4　I/O空間

8086 には，メモリ空間とは別に，I/O 空間という入出力機器とのデータのやりとりに使用される領域が，0000～FFFFH の 64K バイトあります．この空間をアクセスするには，入出力命令と呼ばれる命令を用いなければなりません（図 2-1-5）．I/O 空間をアクセスしてハードウェアを CPU から直接使うプログラムは，第 7 章で説明します．

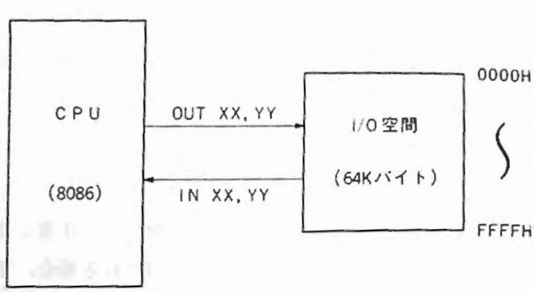


図2-1-5　I/O空間との入出力

2-2　8086の構成

2-2-1　レジスタの分類と役割

8086のレジスタには，図2-2-1に示すようなものがあり，機能的には大きく5つに分類することができます．



図2-2-1　8086のレジスタ構成

①汎用レジスタ

図のAX，BX，CX，DXレジスタは，汎用レジスタと呼ばれ，BASICの変数のように比較的自由に使用することができます．これらはそれぞれ，16ビット長のレジスタとして扱われますが，場合によってはAHとAL，BHとBL，CHとCL，DHとDLというように8ビットレジスタがペアになっているものとしても扱うことができます．ただし，汎用とはいっても無制限ではなく，いくつかの命令では使用されるレジスタが限定されています．

たとえばAX(アキュムレータ)は，乗除算，入出力，ストリング命令などで専用的に使われます．また，演算や転送命令などで使用した時，命令バイト数や実行速度の点で他の汎用レジスタより有利なようになっています．

BX（ベース・レジスタ）は，インダイレクト・アドレッシング（後述）に使用します．

CX（カウント・レジスタ）は，ストリング，繰返し，シフト／ローテイト命令などで，カウンタとして用いることができます．

DX（データ・レジスタ）は，ダブルワードの乗除算命令において上位ワードとして扱われます．

②セグメント・レジスタ

セグメント・レジスタには，図2-2-1で示したような16ビット・レジスタが4本用意されています．これらはCS，DS，SS，ESと呼ばれ，メモリ中のアドレスをアクセスする際には必ずこれらのうちの1つがそのアドレスを決定するためのベースとして用いられます．

CS（コード・セグメント）は，現在実行しているプログラムが属しているセグメントベースを示します．CSの内容は，分岐に関する命令によってのみ意識的に変更することができます．

DS（データ・セグメント）は，主としてプログラムで処理されるデータのあるセグメントを示す場合に使用されます．

ES（エクストラ・セグメント）もDSと同じような用途に使われます．ただ特定の命令（ストリング命令）でのデータ領域の指定には必ずこのESを使用するよう決められています．

SS（スタック・セグメント）は，2-2-2で詳述するスタックと呼ばれる領域のセグメント・ベースを示すものです．

命令によって，これらのセグメント・レジスタのどれが用いられるか，デフォルト（既定値）が決まっていますが，多くの命令でセグメントを変更する命令を挿入することで，他のセグメント・レジスタを指定することができるようになっています．このセグメントの変更をセグメント・オーバーライド（プレフィックス）といいます．

③IP（インストラクション・ポインタ）

マシン語のプログラムは，CSをベースとするコード・セグメント内にあります．この中で，実行される命令の入ったアドレスを示す16ビットのオフセット・レジスタがこのIP（インストラクション・ポインタ）です．つまり実行される命令のあるアドレスは，CSとIPとの組み合せで示されるわけです．

このレジスタの値は，通常プログラムの実行に伴って，アドレスの低位から高位へと順次自動的に更新されます．このレジスタの内容が大きく変更されるのは，分岐を行う命令が実行された時だけです．

④ポインタとオフセット

あるセグメント内でオフセットを指定するために使われるレジスタをポインタといいます．ユーザーの使えるポインタには，BP（ベース・ポインタ）とSP（スタック・ポインタ）があり，SPは，2-2-2で述べるスタックという特殊なデータ処理を実現するために，スタック・セグメントでのアドレッシングに用いられます．BPは，スタック・セグメントにとられたデータ領域をアクセスするためのベース・オフセットとして使われます．

⑤インデクス・レジスタとオフセット

インデクス・レジスタにはSI（ソース・インデクス）とDI（ディスティネーション・インデクス）があります．これらは，通常は任意のデータ・セグメント内のオフセットを表すのに用いられますが，特定の命令（ストリング命令）で使用された場合は，DIがESで指示されたエクストラ・セグメント内のオフセットを示すことになっています．

⑥フラグ

フラグはCPUの状態や動作の制御に使われる16ビットのレジスタで，8086ではその中に各1ビットのフラグが9個，計9ビット分，割り当てられています．フラグの機能は多岐にわたるので，別に項を立て2-2-3で詳しく解説します．

2-2-2　スタックについて

スタックは，いろいろなデータを一時的に保存するために考えられたメモリ中の領域です．しかし，物理的には別に特殊な領域ではなく，8086の1Mバイトのアドレス空間のうちの一部分にすぎません．ただ，この領域を利用する際にある決まったシーケンス（手順）を用いることができ，マシン語プログラムで重要な役割をはたすため，特に説明しておきましょう．

通常メモリ中のあるアドレスは，セグメントとオフセットを使って指定することができます．このことはスタック領域についても例外ではありません．しかしスタックではさらに，図2-2-2に示すように，1つのデータ出入口を通して，いわゆるLIFO（ラストイン・ファーストアウト）の約束でデータをアクセスする方法（命令）がとくに用意されているのです．図では，お皿を箱に入れる時には，上へ上へと重ねていき，取り出すときは上から順に行うことを示しています．皿1，皿2，皿3の順に入れた場合，皿3，皿2，皿1というように取り出すことになります．

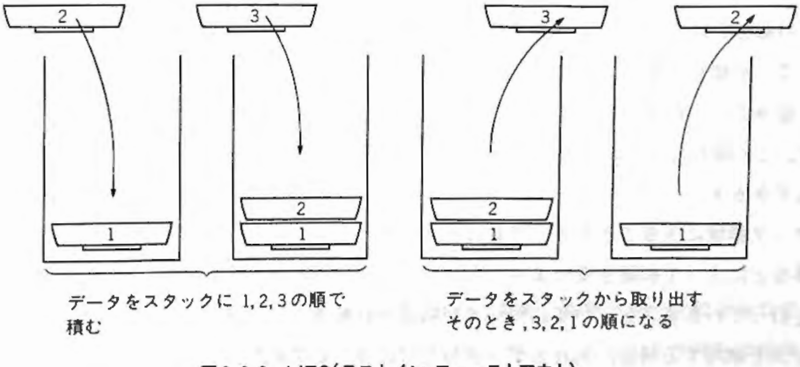


図2-2-2　LIFO(ラストイン・ファーストアウト)

スタックがマシン語プログラムで有効に使われる例として，プログラム中でサブルーチンに分岐する場合をあげることができます．BASICのGOSUB～RETURNを考えていただければわか

るとおり，サブルーチンから戻った時には分岐した次の命令から実行されるように分岐する前の情報を保持しておく必要があります．サブルーチンの入れ子（ネスティング）が行われるなら，その情報はネストに合せて増加します．そして戻る時は保持された情報を逆の順序で取り出してもとのルーチンへ帰ります．そこで，この情報を保持するのにスタックを使用すれば，必要な情報を順序よく出し入れすることができるわけです（図2-2-3）．

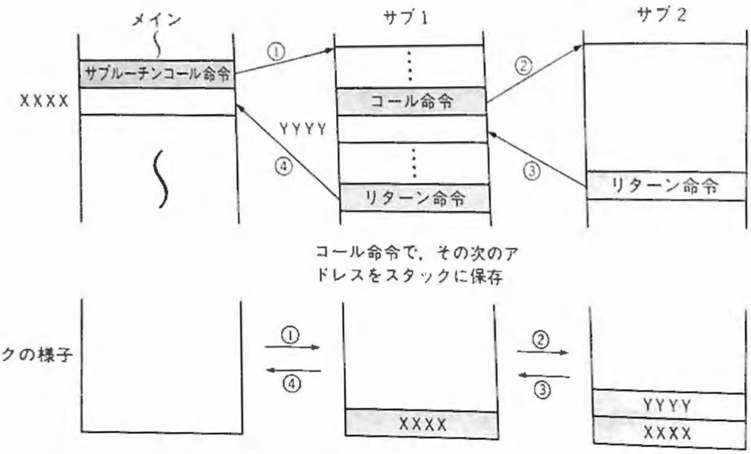


図2-2-3　サブルーチン

このLIFOの実現のために作られたのがスタック領域であり，スタック・ポインタなのです．スタック領域はスタック・セグメント内にとられ，図2-2-4に示すように，スタック領域はメモリの高アドレスから低アドレスに向かってのびていきます．そして，スタック・ポインタは，この領域に最も新しく参照されたデータのアドレス値を保持していて，データを新たに入れる時には，この値を2減少（デクリメント）して，データをストアし，取り出す時にはまずデータを取り出してこの値を2増加（インクリメント）します（16ビット=2バイトデータの場合）．この動作を1命令で行う命令として，PUSH, POPがあります．またサブルーチンのためにCALL, RETがあり，この場合には，コード・セグメントCSの内容も含めてスタックに入れたり出したりすることもできます．

スタック領域に保存できるデータ量はスタック・セグメントのとり方やRAM容量，セグメント境界などによって制限を受けます．ですから，データをスタックに積むだけ，または取り出すだけを行っていきますと，いずれデータが破壊され暴走という状態に陥る可能性が出てきます．スタックを使用する時は，入れたデータ分だけは後で必ず取り出して最終的にはスタック・ポインタの値が元に戻るようにしておく方がよいでしょう（初心者の場合はとくに）．

プログラムの際，スタック領域にデータを保持するようにすると，データを入れた順番に気を付けるだけで，その実際のアドレスを気にする必要がないという利点があります．

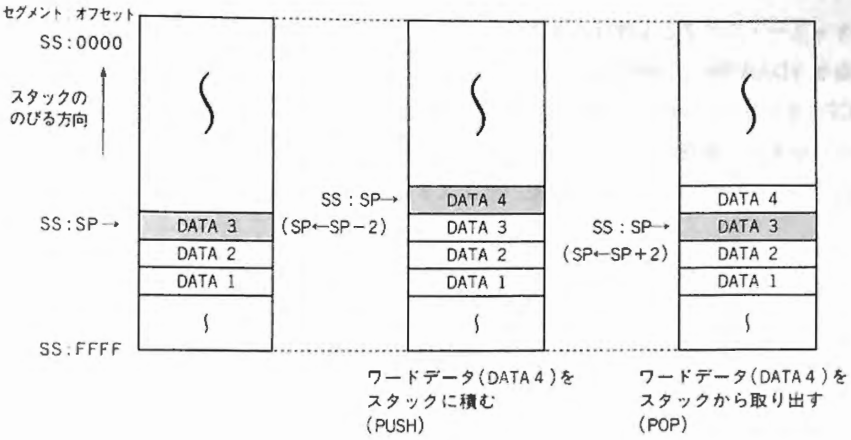


図2-2-4 スタック・ポインタとスタック・セグメント

2-2-3 フラグについて

フラグも，マシン語プログラムにおいて重要な役割をはたします．図2-2-5に示すように，フラグは，1ビットごとに異なる意味をもった9個のビットの集まりで，×印の使われていない部分を含めてフラグ・レジスタまたはFグループと呼ばれます．

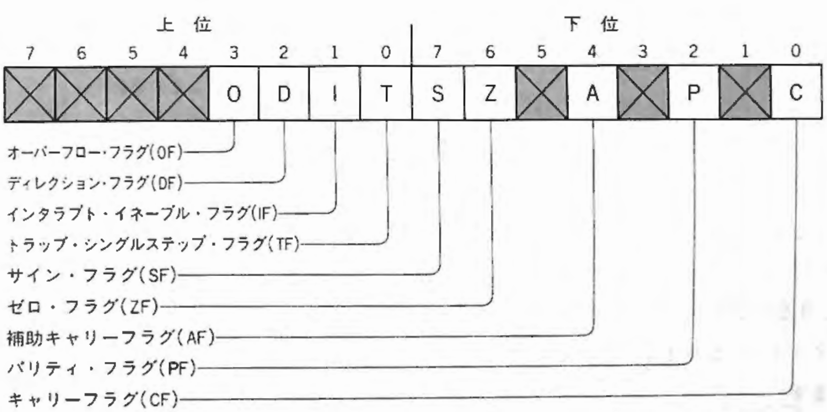


図2-2-5 フラグとフラグ・レジスタ

図中，AF，CF，PF，ZF，OF，SFは，最後に行われた算術，論理演算や比較命令などによって更新されるステータス・フラグで，マシン語プログラムでは，これらのフラグを分岐命令が参照して条件判断を行います．どのフラグがどの命令によってどのように更新されるかは，プログラマーが命令表などを調べて知っていなければなりません．

AF(補助キャリー・フラグ)は，演算後の下位4ビットからの桁上がりを示すフラグで，ハー

フキャリー・フラグとも呼ばれます．分岐命令で直接参照するものはありませんが，10進演算用の命令（DAA等）で参照されます．

CF（キャリー・フラグ）は最上位ビットからの桁上げ（キャリー）や最上位ビットへの借り（ボロー）があった場合にセット（1）され，それ以外はクリアされます．

ZF（ゼロ・フラグ）は演算の結果（ディスティネーション）が0になった場合にセットされます．

PF（パリティ・フラグ）は，ディスティネーション中の1になっているビットの数が偶数であればセットされます．

SF（サイン・フラグ）は，ディスティネーションの最上位ビットのコピーで，0の場合はクリア，1の場合はセットされます．

OF（オーバーフロー・フラグ）は，演算の結果オーバーフローが生じた場合にセットされます．たとえば，8ビットの符号つきの加算において，図2-2-6のように結果が140になってしまった時，これは8ビットの符号つきの数の範囲(−128〜+127)を越えていますからOFはセットされます．

2進数	10進数	
01111000	120	
+) 00010100	+) 20	
10001100	140	………符号なしの場合
	−116	………符号つきの場合

図2-2-6　オーバーフローフラグのセットされる例

残った3つのフラグは，CPUの動作を制御するためのものです．

DF（ディレクション・フラグ）は，ストリング命令（MOVS等）における実行方向を決めるもので，0ならばデータの位置を示すポインタが1増加（インクリメント）し，1ならばその逆（デクリメント）になります．このフラグは，CLD，STD命令により，クリア，セットを行うことができます．

IF（インタラプト・フラグ）は，外部からの割込み信号を受けつけるか，無視するかを決めるフラグです．このフラグは，CLI，STI命令でクリア，セットが行えます．割込みについては，第7章で扱いますので，そちらを参照して下さい．

TF（トラップ・シングルステップ・フラグ）は，CPUをシングルステップ動作させるときに使用するフラグです．

2-2-4　命令の種類とアドレッシングモード

8086の命令は，大きく分けて，**表2-2-1**のように分類することができます．表のニーモニックは，命令の中心部（オペコード部）のみを表しますが，実際の命令は，**図2-2-7**のように3つの形式があります．もっともこれはアセンブリ言語レベルの話であり，マシン語の命令語長（バイト数）との直接の関連はないと考えて下さい．

表2-2-1　命令分類とニーモニック

命　令	ニーモニック
データ転送命令	MOV　PUSH　POP　XCHG IN　OUT　XLAT LEA　LDS　LES LAHF　SAHF　PUSHF　POPF
算術演算命令	ADD　ADC　INC　AAA　DAA SUB　SBB　DEC　NEG　CMP　AAS　DAS MUL　IMUL　AAM DIV　IDIV　AAD　CBW　CWD
論理演算命令	NOT　SHL　SAL　SHR SAR　ROL　ROR　RCL　RCR AND　TEST　OR　XOR
分岐命令	CALL　JMP　RET　JE/JZ　JL/JNGE JLE/JNG　JB/JNAE　JBE/JNA　JP/JPE　JO JS　JNE/JNZ　JNL/JGE　JNLE/JG　JNB/JAE JNBE/JA　JNP/JPO　JNO　JNS　JCXZ LOOP　LOOPZ/LOOPE　LOOPNZ/LOOPNE INT　INTO　IRET
ストリング命令	MOVS　CMPS　SCAS　LODS　STOS REP　REPE/REPZ　REPNE/REPNZ
CPU/フラグ制御命令	CLC　CMC　STC　CLD　STD　CLI　STI HLT　WAIT　ESC　LOCK　NOP

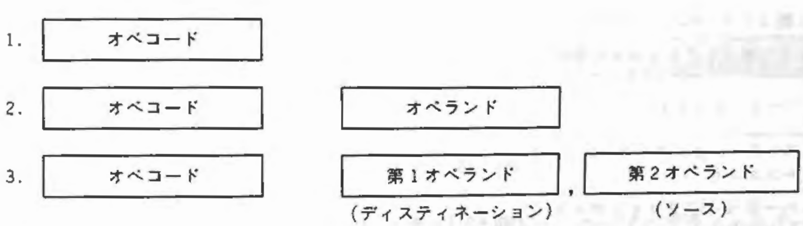


図2-2-7　命令の記述形式

オペランド部は，オペコード部の表す命令のパラメータ部分（処理の対象）を示すもので，2つのオペランドがある場合は，通常第1オペランドがディスティネーションと呼ばれて，処理の結果が残る部分を表し，第2オペランドがソースと呼ばれて，処理の後でも変更されない部分となり

ます．図2-2-8にこの例を示します．

ADD CX,DX	CXとDXの内容を加えてCXの値とする．DX(ソース)の値は変化しない
MOV AX,BX	BXの値をAXに代入する．BXの値は変化しない

図2-2-8　オペランドが2つある場合

マシン語プログラムで，個々の命令の処理対象になるのは，レジスタとメモリや入出力機器のアドレスです．これらの指定方法（アドレッシングモード）は表2-2-2のように分類することができます．

このうち間接モードはさらに表2-2-3に示すバリエーションがあります．

つまり，アドレッシングモードは，オペランド(パラメータ部)の変更によって1つの命令(オペコード)に様々な働きをさせようとすることなのです．これによって命令(オペコード)の種類が整理され，人間にとって多少なりとも理解できる構造になっているわけです．

表2-2-2　アドレッシングモード

	名　称	意　味	例	
1	イミディエイトモード	命令中に直接の値を含む	MOV AX,12	AXに12を入れる
2	レジスタモード	レジスタの指定	MOV BL,CL	CLの値をBLにコピーする
3	ダイレクトモード	対象となる値のアドレス値を命令中に含む	MOV DH,[1000H]*	1000H番地の内容をDHに入れる
4	インダイレクトモード	対象となる値があるアドレスを指定する	MOV DL,[BX]	BXの示すアドレスの内容をDLに入れる

＊MASMでは記述方法が異なる

表2-2-3　インダイレクトモードのいろいろ

インダイレクトモードの種類	オペランド部
ベース・レジスタ	[BX]
ベース・レジスタ＋ディスプレースメント	[BX+d8],[BX+d16],[BP+d8],[BP+d16]
ベース・レジスタ＋インデクス・レジスタ	[BX+SI],[BX+DI],[BP+SI],[BP+DI]
インデクス・レジスタ	[SI],[DI]
インデクス・レジスタ＋ディスプレースメント	[SI+d8],[SI+d16],[DI+d8],[DI+d16]
ベース・レジスタ＋インデクス・レジスタ＋ディスプレースメント	[BX+SI+d8],[BX+SI+d16],[BX+DI+d8],[BX+DI+d16] [BP+SI+d8],[BP+SI+d16],[BP+DI+d8],[BP+DI+d16]

＊インダイレクトモードであることは"[　]"で囲って示す．d8は8ビットの値，d16は16ビットの値を示す

3章 CPU(8086)の基本動作

この章では，8086のマシン語の基本命令をMS-DOSのDEBUG（デバッギング・ツール，"デバッガ"と呼ぶ）を使って簡単なプログラムを実行しながら，第2章で説明したCPUの各レジスタの働き，CPUとメモリ間のデータのやりとりなどを実際に確かめます．また，PC-9801の機械語モニタとの相違点についても解説します．

3-1 マシン語プログラミングの予備知識

3-1-1 なぜDEBUGを使うのか

第1章でも述べたように，MS-DOSのデバッガ(DEBUG．COM)は，マシン語プログラムの動作確認，修正のために，レジスタ内容の表示，変更，プログラムの実行トレース，簡易アセンブラ，逆アセンブラ，メモリ内容の表示，変更などの機能を持っています．これらの機能は，筆者の経験上，入門者がCPUを知り，短いプログラムで命令の動きを学ぶのに最適なツールであるといえます．

ところで，N_{88}-BASIC(86)の機械語モニタモードも，デバッガと同様な機能を持っていますが，この本では後に，MS-DOS上のアセンブラを使用して，本格的なマシン語プログラム開発を行うことを考慮して，デバッガ(DEBUG)を使用することを選びました．なお，機械語モニタモードについては後のデバッガとの相違のところで触れます．

3-1-2 プログラムエリア

さて，マシン語プログラムを実際に作る前に認識しておきたいのがプログラムエリア――つまりメモリ空間のどこにマシン語プログラムをおくかということです．

たとえばBASICでプログラムを作るときに，そのプログラムが，メモリ上のどこからどこまでに記憶されているかを考えながら作る人は少ないでしょう．なぜなら，BASICインタープリタが自動的に適当なメモリ・アドレスを決めてくれるからです．もしもプログラムサイズがメモリ容量を越えてしまうような場合でも，"OUT OF MEMORY" というようにエラー・メッセージを

表示してくれます。

しかし、マシン語プログラムはそう簡単にはいきません。なぜなら、CPU はメモリの有無に関係なく、与えられた命令を順番に、忠実に実行してしまうものだからです。ですから、メモリ上に書いたと思ったプログラムが、途中でメモリがなくなっていたり、データエリアとプログラムエリアが重なっていたりしても、CPU はおかまいなしにアドレスを進めて、ついにはメモリのない、あるいは命令でないアドレスをアクセス（指定し、参照すること）して、わけのわからない動作をしてしまうでしょう。つまり、マシン語プログラムにおいては、BASIC プログラムでは気にもとめなかったメモリ管理という仕事がプログラマーに要求されるのです。

このメモリ管理を行うには、まず PC-9801 のメモリマップを知る必要があります。

メモリマップとは、その名のとおり、CPU のメモリ空間のどこがどのように使われるかを示す地図のことです。図 **3-1-1** に PC-9801 のシステム・メモリマップを示します。

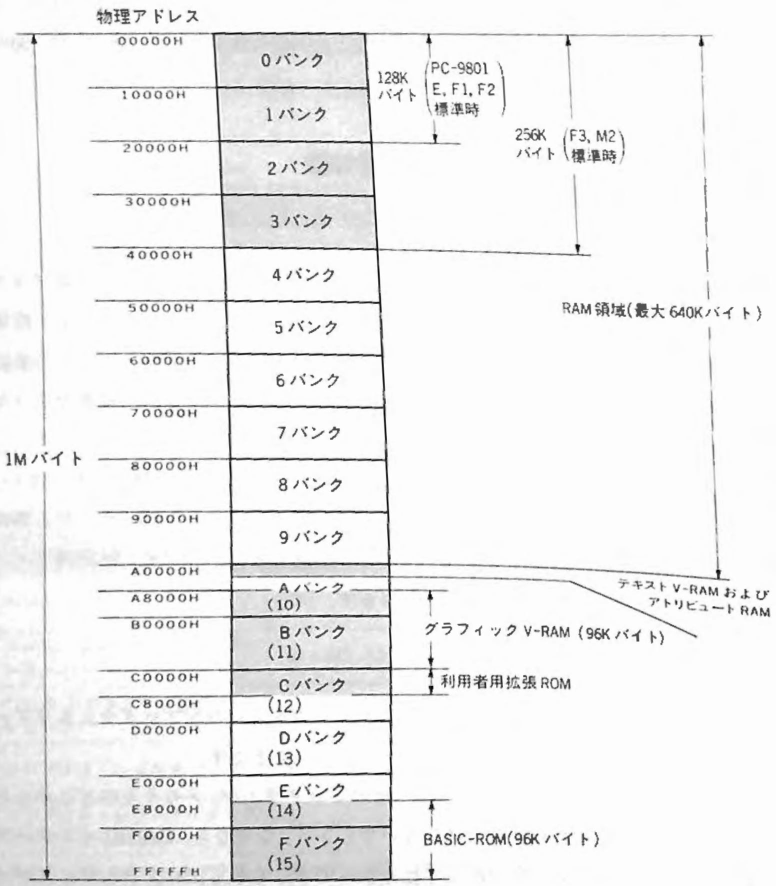


図 3-1-1　PC-9801 のシステム・メモリマップ

図の左に書かれた16進数5桁がメモリの実際のアドレス(物理アドレス)で，8086は1Mバイト（メガバイト）のメモリ空間を持つので，アドレスは00000H～FFFFFHまでとなります．

さて，メモリ空間は64Kバイトごとに0バンクから15バンクまでの16個のバンクに分けられています．メモリ空間の最上位にはN_{88}-BASICインタープリタが納められたROM（リード・オンリー・メモリ）が96Kバイトを占め，その下にシステムのための領域が続き，A8000Hからグラフィック V-RAMがあることがわかります．そして00000Hから9FFFFHまでの640Kバイトが RAMのエリアになっています．しかし，標準では128Kバイトつまり0バンクと1バンクにしか RAMは実装されていません(PC-9801 F3，Mは256Kバイト実装)．したがって基本的には，この128Kバイトのメモリ空間内だけが，ユーザーが作るマシン語プログラムの対象となるわけです．

とはいうものの，デバッガ使用時にはMS-DOSの本体とデバッガ自身もメモリ上になくてはいけませんので，実際にユーザーがプログラムを書くことのできるエリアは，図3-1-2のようになります．このエリアは，RAMを増設することで拡張できます．

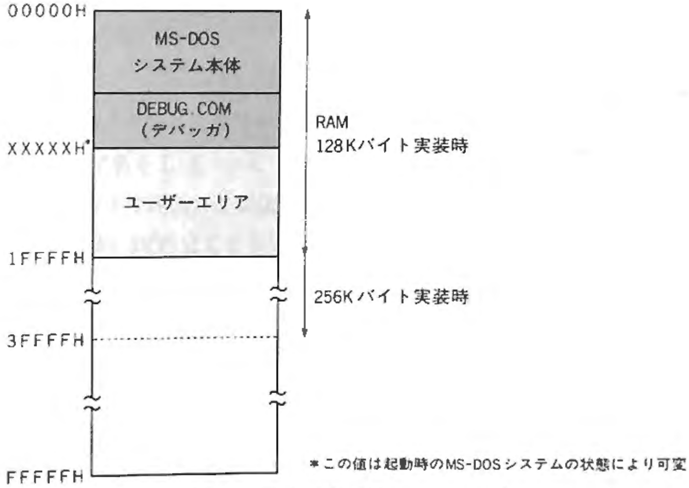


図3-1-2　デバッガ使用時のメモリマップ

以上のように，マシン語プログラミングをするうえで対象となるのは，RAM上の限られたメモリ空間であり，そのアドレスにはプログラマーが十分気をつけなくてはいけないことを認識してください．

3-1-3　アセンブリ言語の表現形式

デバッガでマシン語プログラミングを行ううえで必要なもう1つの予備知識は，アセンブリ言語の表現形式です．これは，8086のアセンブリ言語で，内部レジスタや，メモリ・アドレス，メモリ内容をどのように表現するかということで，ここでは，デバッガのワンライン・アセンブラに関するものを説明します．

○レジスタ名

レジスタ名は，表3-1-1のように表現します．

表3-1-1 レジスタの表現

レジスタの種類	デバッガ，アセンブラでの表現			
セグメント・レジスタ	CS	DS	ES	SS
汎用レジスタ(16ビット時)	AX	BX	CX	DX
汎用レジスタ(8ビット時)	AH, AL	BH, BL	CH, CL	DH, DL
インデックス・レジスタ	SI	DI		
ポインタ・レジスタ	BP	SP	IP(PC)	
フラグ・レジスタ	F (Rコマンド，フラグ・レジスタの指定に使用)			

○メモリ・アドレスとメモリ内容

8086は，1Mバイトのメモリ空間の1つのアドレスを指定するためにセグメントとオフセットを使います．これは，

セグメント値：オフセット値

を16進数の形で書き表し，例えば，

2000：10F0

ならば，実際のアドレスは，セグメント値×16＋オフセット値，つまり，

20000H＋10F0H＝210F0H

ということになります．16進数は，このように最後にHを付けて表すことにします．

また，セグメント値をセグメント・レジスタ名にすれば，そのセグメント・レジスタの内容をセグメント値とした実アドレスを指すことになります．つまり，

CSの内容＝2000Hのとき，
CS：10F0＝210F0

となります．このとき，CS：，DS：，ES：，SS：のことをセグメント・オーバーライド・プレフィックスと呼んでいます．このようにして指定したメモリ・アドレスの内容を示すには，このアドレスに角カッコ"[ ]"をつけて表します．プログラム中では，

[オフセット値]（またはセグメント名：[オフセット値]）

のように表します．たとえば，アドレス1E500H番地の内容がA0Hであるとして，CSのセグメント値が1000Hだとすると，

[E500]（または CS：[E500]）＝A0H

ということになります．

以上は，ごく基本的なことですが，よく確認してください．その他のことは，実際に出てきたところで説明していきたいと思います．

3-1-4 DEBUGの起動

では，デバッガを起動してみましょう．

PC-9801用のMS-DOSシステムディスクをドライブ1に入れ，リセットを行うと，MS-DOSが起動します．ここで，

A＞DEBUG⏎

と入力すると，

－

と表示され，デバッガのコマンド入力待ちになります．

BASICでは，ドライブ名を1, 2,……，と番号で呼んでいましたが，MS-DOSでは，A，B，C,……，とアルファベットで呼ぶことに注意してください．

さて，デバッガからMS-DOSにもどるには，

－Q⏎

とします．これは，Quit（終了）コマンドです．

3-2 8086の動作を見る

3-2-1 プログラミングの準備

デバッガが起動したら，まず"R"コマンドを入力してみます．コマンドは，大文字でも小文字でもかまいません．このコマンドは，Register（レジスタ）コマンドといって，結果は，図3-2-1のように，現在のレジスタの内容を表示します．

```
-R
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=11BC  ES=11BC  SS=11BC  CS=11BC  IP=0100  NV UP DI PL NZ NA PO NC
11BC:0100 0807          OR      [BX],AL                          DS:0000=CD
-
```

フラグの内容（NV UP DI PL NZ NA PO NC）

現在指している命令の内容（OR [BX],AL）

図3-2-1 Rコマンドによる表示

ここに表示されたのは，デバッガ起動時の全レジスタの初期状態です．このように，AX，BX，CX，DXの各レジスタの値は，0であり，全セグメント・レジスタは，11BCHにセットされています．この値は，デバッガが用意したフリーエリアを示しており，すべてのセグメント・レジスタが同じであることから，プログラムもデータも，64Kバイトの同一セグメント内にあることになります．

また，IP（インストラクション・ポインタ）は，オフセット値0100Hを示しており，デバッガに入った状態では命令実行は，セグメント11BCH，オフセット0100H（11CC0H）から準備されていることがわかります．

今後は，この状態をもとにプログラムを作ることにします．つまり，セグメント・レジスタはすべて同一にし，プログラムエリアはオフセット値100Hから，データはそのあとにとることにします（図3-2-2）．

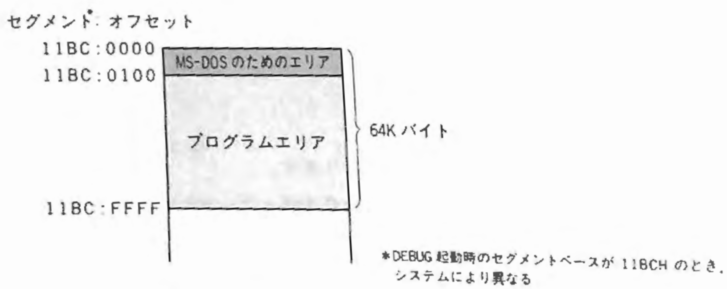


図3-2-2　プログラムエリアのメモリマップ

セグメント・レジスタの値（ここでは11BCH）はメモリの搭載量やシステムの状態によって変化しますので，ここではあまり気にする必要はありません．ただ，オフセットの値は変えないよう注意してください．

次にメモリの内容を調べてみましょう．Dコマンドを入力してください．このコマンドはDump（ダンプ）コマンドといい，結果は図3-2-3のようにセグメント値：オフセット値，メモリ内容（16進数），メモリ内容に対応したアスキー文字の順で表示されます．

```
-D
11BC:0100  08 07 50 52 4F 4D 50 54-00 AE 06 05 50 41 54 48   ..PROMPT....PATH
11BC:0110  00 27 1A 05 45 58 49 54-00 BF 0A 05 43 54 54 59   . ..EXIT.?..CTTY
11BC:0120  01 DA 0A 05 45 43 48 4F-00 B3 18 05 47 4F 54 4F   .Z..ECHO.3..GOTO
11BC:0130  00 53 0E 06 53 48 49 46-54 00 12 0E 03 49 46 00   .S..SHIFT....IF.
11BC:0140  0C 0D 04 46 4F 52 00 23-0C 04 43 4C 53 00 F3 12   ...FOR.#..CLS.s.
11BC:0150  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
11BC:0160  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
11BC:0170  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
-
```

アドレス（セグメント値：オフセット値）　メモリ内容の16進表示　メモリ内容に対応したアスキー文字

図3-2-3　Dコマンドによる表示

表示される範囲は，ここでは 11BC：0100H から 11BC：017FH までの 128 バイトです．この先頭アドレスは，先ほど述べたように，デバッガが用意したユーザー用のフリーエリアの先頭で，ここから後のメモリを，ユーザーは自由に使用できるわけですが，ダンプの結果では何やらデータが入っています．これは MS-DOS の起動にともなって生じた不要のゴミです．これから，この領域にプログラムを作っていきますので，まずこのゴミを消してしまいましょう．それには F (Fill) コマンドを使います．

```
-F  100  2FF  0⏎
```

こうすると，オフセットの 100H～2FFH 番地のメモリを 00H でうめることになります．

D コマンドで確かめるには，

```
-D  100  2FF⏎
```

とします．これは，オフセット 100H～2FFH 番地までのメモリ内容をダンプする命令です．この結果を図 3-2-4 に示します．

```
-F 100 2FF 00
-D 100 2FF
11BC:0100  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
11BC:0110  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
11BC:0120  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
11BC:0130  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
11BC:0140  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
11BC:0150  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
11BC:0160  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
11BC:0170  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
11BC:0180  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
11BC:0190  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
11BC:01A0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
11BC:01B0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
11BC:01C0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
11BC:01D0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
11BC:01E0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
11BC:01F0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
11BC:0200  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
11BC:0210  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
11BC:0220  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
11BC:0230  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
11BC:0240  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
11BC:0250  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
11BC:0260  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
11BC:0270  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
11BC:0280  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
11BC:0290  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
11BC:02A0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
11BC:02B0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
11BC:02C0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
11BC:02D0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
11BC:02E0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
11BC:02F0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
-
```

図3-2-4　F コマンドの結果

3-2-2 プログラムの入力

ここでは、CPUの動作を知ってもらい，マシン語プログラミングのフィーリングをつかんでもらうため，簡単なプログラムを使います．そのプログラムは次のとおりです（図3-2-5）．

```
アドレス
0100H  MOV  AX, 1234      ┐ 1234Hを200,201H番地に格納する
0103H  MOV  [200], AX     ┘
0106H  MOV  BX, 5678      ┐ 5678Hを202,203H番地に格納する
0109H  MOV  [202], BX     ┘
010DH  CMP  AX, BX   ── AXとBXの値を比較する
010FH  JA   112      ── AX>BXなら112H番地へ，そうでなければ次の命令へ行く
0111H  XCHG AX, BX   ── AXとBXの値を入れ換える
0112H  SUB  AX, BX   ── AXからBXの値をひく．結果はAXに入る
0114H  MOV  [204], AX ── AXの値を204H番地に格納する
```

図3-2-5 マシン語サンプル・プログラム

このプログラムは，サンプルのために作ったもので，1234Hと，5678Hという値をAX，BXレジスタに格納してから，メモリの200H番地，202H番地からにそれぞれ書き込み，レジスタの値を比較してAXがBXより大きければ112H番地に，そうでなければAX，BXの値を入れかえてから112H番地に行き，AXからBXの値を引いて，204H番地に書き込んで終了するものです．

これを入力するには，まずデバッガの"A"コマンドを使用します．これはAssemble（アセンブル）コマンドといい，ニーモニックを入力すると，それに対応したマシン語に変換し，指定されたアドレスのメモリ上に書き込みます．

−A⏎

"A"を入力すると，現在着目しているアドレスが表示されるので，ここにニーモニックを書き，リターンキーを押します（図3-2-6）．

```
-A 100
11BC:0100 MOV AX,1234
11BC:0103
```

図3-2-6 Aコマンドによるプログラムの入力

すると次のアドレスが表示されるので，同様にして入力します．入力が終わったら，リターンのみを入力するか，CTRL+Cを入力すればコマンド待ち状態にもどります．

さて，今のプログラムが正しく記憶されているかどうかを調べるには，"U"（Unassemble）コマンドを使います．それには，

−U 100⏎

のように入力します．すると32バイト目までの命令が図3-2-7のように表示されます．

```
-U 100
11BC:0100 B83412        MOV     AX,1234      ┐
11BC:0103 A30002        MOV     [0200],AX    │
11BC:0106 BB7856        MOV     BX,5678      │
11BC:0109 891E0202      MOV     [0202],BX    │ プログラムの内容
11BC:010D 39D8          CMP     AX,BX        │
11BC:010F 7701          JA      0112         │
11BC:0111 93            XCHG    BX,AX        │
11BC:0112 29D8          SUB     AX,BX        │
11BC:0114 A30402        MOV     [0204],AX    ┘
11BC:0117 0000          ADD     [BX+SI],AL
11BC:0119 0000          ADD     [BX+SI],AL
11BC:011B 0000          ADD     [BX+SI],AL
11BC:011D 0000          ADD     [BX+SI],AL
11BC:011F 0000          ADD     [BX+SI],AL
-
```

図3-2-7　Uコマンドによるプログラムの表示

これは，さきほど入力したプログラムの開始アドレスのオフセット値100Hからメモリに記憶されている16進数を，対応するニーモニックに変換して表示するという意味で，この結果から，先ほどのプログラムは，17H（16進）つまり23バイトのマシン語になっていることがわかります．

図3-2-7のプログラムの逆アセンブルリストで，第2章で説明したワードデータ(2バイトデータ)の格納のされ方を確かめることができます．たとえば，1行目のマシン語とニーモニックを比べると，

マシン語	ニーモニック（アセンブリ言語表記）
B83412	MOV　AX，1234

となっています．

ここでマシン語のB8Hが示すのは，

MOV　AX，<2バイト定数>

という形の命令で，これは，「<2バイト定数>をAXレジスタに移す」という意味です．するとその次の"3412"というのが，ニーモニックの1234Hに対応するわけです．これを見ると，ワードデータが，上下がひっくり返って格納されていることが確認できます．

3-2-3 プログラムの格納と呼出し

プログラムの入力が終わったからといってすぐに実行するのは考えものです．なぜなら，マシン語プログラムは，万一バグがあると，最悪の場合，そのプログラム自身はおろか，メモリ上のMS-DOSも破壊されてしまうことすらあるからです．これは，永久的な破壊ではないので，もう一度システムを起動すればよいのですが，せっかく入力したプログラムを，もう一度入れ直さなくてはなりません．

そこで，これをディスクに格納しておきましょう．それにはまず，"N"（Name）コマンドで，このプログラムを何という名前のファイルにするかを指定します．ここでは"CH3－1"という名前を使用します．

```
-N CH3-1↵
```

次に，プログラムをディスクに書き込むには，まずプログラムの総バイト数をCPUのBX，CXレジスタに書いておきます．これにはRコマンドを使います．

このとき，BX，CX各16ビットのレジスタが上位，下位の順でつなげられて，32ビットのレジスタとして使用され，理論的には 2^{32}＝FFFFFFFFH (4294267296) バイトまでの大きさを示すことができます．もちろん，PC-9801のメモリ領域は1Mバイト以内ですので，このような長大なプログラムを書くことはできませんが，64Kバイト以上のプログラムの格納の時には威力を発揮します．たとえば，20000H（＝131072）バイトのプログラムなら，

```
BX=0002H，CX=0000H
```

とすればよいのです．

今は，格納するべきプログラムのサイズは17H(23)バイトですので，BXとCXは，それぞれ，

```
BX=0000H，CX=0017H
```

とすればよいでしょう．これには，

```
-RBX↵
```

とします．すると，

```
BX 0000
:
```

と表示され，BXが0であることがわかります．変更の必要はないのでリターンを入力し，こんどは，

```
−RCX⏎
CX　0000
：17⏎
```

と入力します．これでレジスタに値がセットされました．そして "W"(Write)コマンドを使って，

```
−W⏎
```

とすれば，書き込みが行なわれます（図 3-2-8）．

```
-N CH3-1 ──────── ファイル名の指定
-R BX    ──────── BXレジスタの確認
BX 0000
:
-R CX    ──────── CXレジスタ内容の変更
CX 0000
:17
-W       ──────── ファイルへの書き出し
Writing 0017 bytes
-
```

図 3-2-8　プログラムのセーブ

次に格納したプログラムをメモリ上にもってくるには "L" (Load) コマンドを使います．これには，書込みと同様に N コマンドで名前を指定し，次に L コマンドを実行します(図 3-2-9)．

```
-N CH3-1
-L
-U 100
11BC:0100 B83412        MOV     AX,1234
11BC:0103 A30002        MOV     [0200],AX
11BC:0106 BB7856        MOV     BX,5678
11BC:0109 891E0202      MOV     [0202],BX
11BC:010D 39D8          CMP     AX,BX
11BC:010F 7701          JA      0112
11BC:0111 93            XCHG    BX,AX
11BC:0112 29D8          SUB     AX,BX
11BC:0114 A30402        MOV     [0204],AX
11BC:0117 0000          ADD     [BX+SI],AL
11BC:0119 0000          ADD     [BX+SI],AL
11BC:011B 0000          ADD     [BX+SI],AL
11BC:011D 0000          ADD     [BX+SI],AL
11BC:011F 0000          ADD     [BX+SI],AL
-R
AX=0000  BX=0000  CX=0017  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=11BC  ES=11BC  SS=11BC  CS=11BC  IP=0100   NV UP DI PL NZ NA PO NC
11BC:0100 B83412        MOV     AX,1234
-                               └── プログラムのバイト数(16進数)
```

図 3-2-9　プログラムのロードと確認

この例では，Lのあとに何も指定しませんので，プログラムは，コード・セグメントのオフセット100Hからロードされます．別のアドレスにもっていきたい場合には，

L<アドレス>

のように指定します．ただし，ここで作ったプログラムは，オフセット100Hから格納されていないと正しく動きません．なお，Lコマンド終了時はBX，CXレジスタに，今ロードしたプログラムのサイズが入ります．

ここで一度MS-DOSにもどってディレクトリをみてみましょう（図3-2-10）．

```
-Q

A>DIR /W

 ドライブ A: のディスクにはボリュームラベルがありません
 ディレクトリは A:¥

COMMAND  COM    CHKDSK   COM    KNJDIC   SYS    DEBUG    COM    EDLIN    COM
MORE     COM    RECOVER  COM    SYS      COM    ASSIGN   COM    COPY2    COM
COPYA    COM    DICM     COM    DISKCOPY COM    DUMP     COM    FORMAT   COM
KEY      COM    PRINT    COM    SPEED    COM    SWITCH   COM    USKCGM   COM
CREF     EXE    EXE2BIN  EXE    FC       EXE    FIND     EXE    LIB      EXE
LINK     EXE    MASM     EXE    SORT     EXE    CONFIG   SYS    CH3-1
         30 個のファイルがあります                                  ↑
    836608 バイトが使用可能です                              格納したプログラム
```

図3-2-10　ディレクトリの表示

正しくファイルができているのがわかります．

プログラムのロードには，先にデバッガに入ってからLコマンドを使う方法がありましたが，デバッガの起動時に指定して，

A>DEBUG　CH3−1

のようにすることで，オフセット100Hからロードすることができます．このとき，レジスタの内容は，CXレジスタのみが0017Hと変っています．

3-2-4　プログラムの実行

では，プログラムを実行してみましょう．前述したように，このプログラムは，アキュムレータAXとBXに2バイトの値を入れて，それぞれをメモリに格納し，お互いを比較して大きい方から小さい方を引き，結果をメモリにしまうものです．それぞれの値を格納するメモリのアドレスは，

AXの内容　　―― 200H番地（2バイト）
BXの内容　　―― 202H番地（2バイト）
AXとBXの差　―― 204H番地（2バイト）

となります．

そして，アキュムレータに入れるデータはそれぞれ，

AX＝1234H，BX＝5678H

です．

実行に先立って結果の格納されるメモリの内容をDコマンドで確かめます（図3-2-11）．

```
-D 200 20F
11BC:0200  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
-
```

図3-2-11　データ領域の表示

すべて0で埋められていることが確認されたら，実行してみましょう．

プログラムの実行はG (Go) コマンドで行います．Gコマンドは，実行を始める開始番地と，途中で止めてデバッガにもどる点（ブレーク・ポイント）を指定します．この例のプログラムには，終了のための命令がないため，ブレーク・ポイントを設定しておかないと暴走してしまいます．

ここでは，CSからのオフセット100H番地から114H番地までの命令を実行させるので，ブレーク・ポイントは次の117H番地に設定します．

−G=100　117⏎

すると，すぐにレジスタの内容が表示され，デバッガのコマンド待ち状態になります．ここで結果の格納されたメモリをダンプしてみます（図3-2-12）．

```
            ┌──ブレークポイント・アドレス(このアドレスの命令は実行されない)
-G=100 117
     └──実行開始アドレス
AX=4444  BX=1234  CX=0017  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=11BC  ES=11BC  SS=11BC  CS=11BC  IP=0117   NV UP DI PL NZ NA PE NC
11BC:0117 0000          ADD     [BX+SI],AL                       DS:1234=00
-D 200 20F
11BC:0200  34 12 78 56 44 44 00 00-00 00 00 00 00 00 00 00   4.xVDD..........
-          AXの値 BXの値 演算結果
           (上位,下位が逆に入っていることに注意)
```

図3-2-12　Gコマンドによるプログラムの実行

さて，結果はやはり上下バイトが逆さまですが，各メモリには予定通り，

1234H，5678H，4444H

が順に格納されていました．

3-2-5 メモリ内容の変更

先ほどのプログラムでは，BXレジスタの内容がAXレジスタの内容よりも大きかったわけですが，逆ではどうでしょうか．次は，

AX＝4567H，BX＝2345H

としてみましょう．結果は2222Hのはずです．

このように変更するには，Aコマンドでプログラムの一部を書き換えればよいでしょう．1行目を変えるには，

```
－A　100↵
11BC：100　MOV　AX，4567↵
11BC：103↵
－↵
```

とすればできます．

もう1つ別の方法があります．それは"E"(Enter)コマンドを使って1バイトずつ直接メモリを書き換える方法です．これはデータを変更する時には便利ですが，ワードデータは，上下の順に気をつけなくてはなりません．3行目の5678Hを2345Hに変えてみます．これにはまず，

```
－E　106↵
```

と入力します．すると，

```
11BC：0106　BB．
```

と表示されます．ここで16進数を2桁入力すればこのメモリ内容は変更されますが，ここでは変更の必要がないのでスペースキーを一度押します．すると次のアドレスの内容が表示されます．

```
11BC：0106　BB．　78．
```

これは，5678Hの78なので，45を入力してスペースキーを押します．

```
11BC：0106　BB．　78．45
11BC：0108　56．
```

すると次のアドレスが表示されます．同様にして今度は，23を入力してリターンすれば，変更は終了です．Uコマンドで正しく変更されたかどうか確かめてください（図3-2-13）．

```
-U 100 114
11BC:0100 B83412        MOV     AX,1234
11BC:0103 A30002        MOV     [0200],AX
11BC:0106 BB7856        MOV     BX,5678
11BC:0109 891E0202      MOV     [0202],BX
11BC:010D 39D8          CMP     AX,BX
11BC:010F 7701          JA      0112
11BC:0111 93            XCHG    BX,AX
11BC:0112 29D8          SUB     AX,BX
11BC:0114 A30402        MOV     [0204],AX
-A 100
11BC:0100 MOV AX,4567 ──────────────── Aコマンドによるプログラムの変更
11BC:0103
-E 106
11BC:0106  BB.      78.45 ──────────── Eコマンドによるプログラムの変更
11BC:0108  56.23
-U 100 114
11BC:0100 B86745        MOV     AX,4567 ─┬─ 変更の確認
11BC:0103 A30002        MOV     [0200],AX│
11BC:0106 BB4523        MOV     BX,2345 ─┘
11BC:0109 891E0202      MOV     [0202],BX
11BC:010D 39D8          CMP     AX,BX
11BC:010F 7701          JA      0112
11BC:0111 93            XCHG    BX,AX
11BC:0112 29D8          SUB     AX,BX
11BC:0114 A30402        MOV     [0204],AX
-E 200
11BC:0200  34.0     12.0     78.0     56.0     44.0     44.0     00.── Eコマンドでメモリを
-D 200 20F                                                            クリアする
11BC:0200  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
-
```

図 3-2-13 プログラムの変更

なお，図 3-2-13 では，同じく E コマンドで，結果の格納領域である 0200H～0205H 番地までを 0H に変更する様子も示しておきます．

さて，ここで再び実行後，結果をみてみると，図 3-1-14 のようになります．このプログラムは，"CH3－2" という名前でセーブしておきます．

```
-G=100 117 ────────── プログラムの実行

AX=2222  BX=2345  CX=0017  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=11BC  ES=11BC  SS=11BC  CS=11BC  IP=0117   NV UP DI PL NZ NA PE NC
11BC:0117 0000          ADD     [BX+SI],AL                       DS:2345=00
-D 200 20F
11BC:0200 [67 45] [45 23] [22 22] 00 00-00 00 00 00 00 00 00 00   gEE#""..........
-N CH3-2─┐ AXの値   BXの値  演算結果
-R BX    │
BX 2345  └──────── ファイルの指定
:0 ──────────────── BXレジスタの値を0にセット
-R CX
CX 0017
: ───────────────── CXレジスタの値は17H なので変更なし
-W ──────────────── ファイルへの格納
Writing 0017 bytes
-
```

図 3-2-14 プログラムの実行とセーブ

3-2-6 実行のトレース

これまでのオペレーションで例題のプログラムは正しく動くことはわかりました．では，CPUはどのようにしてこのプログラムを実行しているのでしょうか．

デバッガには，マシン語プログラムを1命令ずつ実行し，その都度レジスタの状態を表示する"T"(Trace) コマンドがあります．これを使ってCPUの基本動作とデータの動き，処理のされ方をみてみましょう．

まず，100Hから2FFH番地までをFコマンドで0にします．次に，LコマンドでCH3−1プログラムをロードします．さらにRコマンドで，BX，CXの値を0にします．また，IPをプログラムの先頭番地である100Hにします（図3-2-15）．

```
-F 100 2FF 00
-N CH3-1 ──────── CH3-1プログラムの再ロード
-L
-R
AX=0000  BX=0000  CX=0017  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=11BC  ES=11BC  SS=11BC  CS=11BC  IP=0117   NV UP DI PL NZ NA PE NC
11BC:0117 0000          ADD     [BX+SI],AL                    DS:0000=CD
-R CX
CX 0017 ──────── CXレジスタのクリア
:0
-R
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=11BC  ES=11BC  SS=11BC  CS=11BC  IP=0117   NV UP DI PL NZ NA PE NC
11BC:0117 0000          ADD     [BX+SI],AL                    DS:0000=CD
-R IP
IP 0117
:100 ──── IP(インストラクション・ポインタ)のセット
-
```

図3-2-15　プログラムのロード

では，最初にRコマンドで現在のプログラム実行前のレジスタの状態を確かめます(図3-2-16)．

```
-R
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=11BC  ES=11BC  SS=11BC  CS=11BC  IP=0100   NV UP DI PL NZ NA PE NC
11BC:0100 B83412        MOV     AX,1234
-
```

図3-2-16　レジスタ内容の表示

ここでRコマンドの結果をくわしく明記すると，AX＝XXXXのように，"レジスタ名＝<16進4桁>"は，それぞれのレジスタの内容を示しています．ここでは，4つのセグメント・レジスタ(DS，ES，SS，CS）の値は11BCH（起動時の値，変動あり），SP（スタック・ポインタ）の値はFFEEH，そしてIP（インストラクション・ポインタ）の値は0100Hで，それ以外のレジスタは0になっています．

IPの後の2文字のアルファベットの列は，フラグ・レジスタの状態を示しています．そして次の行には，現在着目しているアドレス，つまりIPの指しているアドレスとその内容がマシン語とニ

ーモニックで表示されています．今はプログラムの先頭です．

これらのレジスタの値は，直前に実行された命令の結果になるのですが，今はRコマンドで初期化したものになっています．

では，Tコマンドで1ラインずつ命令を実行していきましょう．それには，

－T＝100⏎

と入力します．これで100H番地からトレースを行うことを示します．

トレースの結果は，図3-2-17のようにRコマンドの結果と同じ表示がなされますが，各レジスタの値は，プログラムの1行目の結果により変わっています．

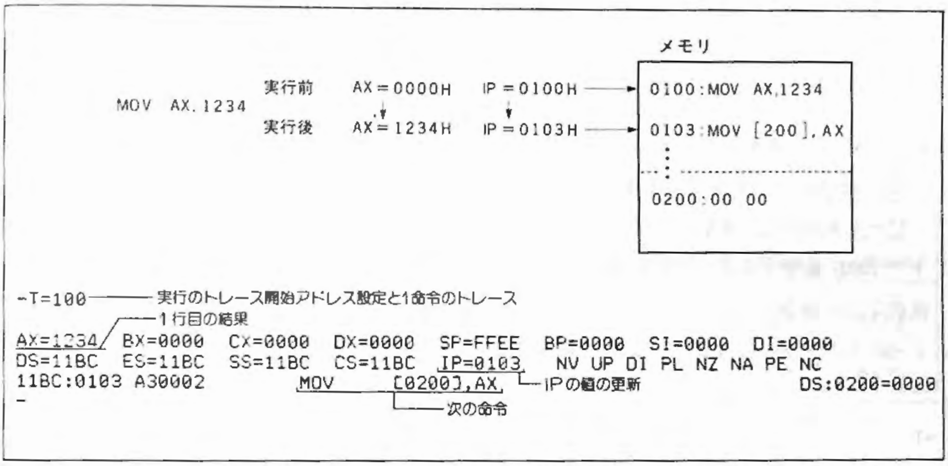


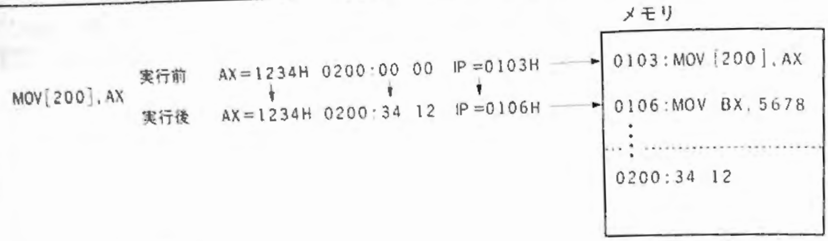
図3-2-17　命令の実行によるレジスタ内容の変化(1)

今実行したのは，

MOV　AX，1234H

で，これはAXレジスタに定数1234Hを入れるというMOV（ムーブ）命令の1つです．このような命令はデータ転送命令といわれ，これを解釈したCPUは，プログラム中の定数1234HをAXレジスタにコピーします．この結果，図3-2-17のように，AX＝1234と変わりました．さらに，CPUは次の命令を読むため，IPを今行った命令のバイト数だけ増やして次の命令のアドレスをさします．この結果，次の着目点のアドレスとその内容が表示されています．

次の命令は，同じMOV命令ですが，今度はAXレジスタの内容を200H番地のメモリに格納します．ここでアドレスに角カッコがついているのは，この命令が，AXレジスタの内容を200H番地にコピーするという意味です．これを実行するには単にTを入力します．これで次の命令を1つだけ実行し，その結果が表示されます（図3-2-18）．




```
-T ——————— 1命令の実行

AX=1234  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=11BC  ES=11BC  SS=11BC  CS=11BC  IP=0106   NV UP DI PL NZ NA PE NC
11BC:0106 BB7856        MOV     BX,5678
-D 200 20F
11BC:0200  34 12 00 00 00 00 00 00-00 00 00 00 00 00 00 00   4...............
-          └── 2行目の実行結果
```

図3-2-18　命令の実行によるレジスタ内容の変化(2)

結果を見ても，変わっているのはIPの値と次の命令だけで他には何の変化もありません．しかし，CPUはAXの内容を200H番地からの2バイトのメモリにコピーしています．これはあくまでコピーであるので，AXの内容は依然として1234Hであることに気をつけてください．Dコマンドで200H番地をみると正しく実行されたことがわかります．

次の2つの命令は，上で述べたのと同様に，BXレジスタに5678Hを移し，BXを201H番地から2バイトのメモリにコピーします（図3-2-19）．

```
-T
                 ┌──BXへの値のセット
AX=1234  BX=5678/ CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=11BC  ES=11BC  SS=11BC  CS=11BC  IP=0109   NV UP DI PL NZ NA PE NC
11BC:0109 891E0202      MOV     [0202],BX
-T                                                            DS:0202=0000
AX=1234  BX=5678  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=11BC  ES=11BC  SS=11BC  CS=11BC  IP=010D   NV UP DI PL NZ NA PE NC
11BC:010D 39D8          CMP     AX,BX
-
```

図3-2-19　命令の実行によるレジスタ内容の変化(3)

これでAX，BXにそれぞれ値が入りました．次は10DH番地の命令です．

CMP　AX，BX

これは，AXとBXの内容を比較するCMP(Compare)命令で，実際にはCPUはAXからBXを減算して，その結果でフラグが変化します．図3-2-20が実行結果ですが，減算を行ってもAX，BXともに変化はありませんが，フラグが3カ所変わっています．つまり，CMP命令は比較するものは不変とし，結果をフラグの変化だけで残すというわけです．

```
-T

AX=1234  BX=5678  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=11BC  ES=11BC  SS=11BC  CS=11BC  IP=010F   NV UP DI NG NZ AC PO CY
11BC:010F 7701          JA      0112                   └──┴─CMP命令の結果
-
```

図3-2-20 命令の実行によるレジスタ内容の変化(4)

ここでAXとBXを比較したのは，AXとBXのうち大きい方から小さい方を引いて差を求めるためで，今は当然BXの方が大きいという結果になります．

3-2-7 フラグの変化を見る

では，それぞれのフラグはどんな意味をもっているのでしょうか．デバッガが表示するフラグ・レジスタの内容は図3-2-21のようになっており，実際はCPU内部で1，0で表されるフラグの内容を，わかりやすいように2文字の略号で表します．

フラグ名	Overflow (オーバーフロー 桁あふれ)	Direction (ディレクション 方 向)	Interrupt (インタラプト 割込み)	Sign (サイン 符 号)	Zero (ゼロ 零判定)	Auxiliary Carry (オークシアリーキャリー 補助キャリー)	Parity (パリティ 検査符号)	Carry (キャリー 桁上げ)
ビット	OV	DN (減 少)	EI (許 可)	NG (負)	ZR (結果が0)	AC	PE (偶)	CY (桁上げあり)
クリア	NV	UP (増 加)	DI (不許可)	PL (正)	NZ (0でない)	NA	PO (奇)	NC (桁上げなし)

例) CMP AX,BX(AX＝1234H,BX＝5678H)の結果
フラグの表示) NV UP DI NG NZ AC PO CY
意味) NV : オーバーフロー(符号つきデータでのオーバーフロー)なし
UP : ディレクション(LOOP命令などでのカウンタの増減)は増加方向
DI : 割込みは不許可
NG : 処理結果の符号は負
NZ : 処理結果は0ではない
AC : 補助キャリー(BCD演算用)はセット
PO : パリティ(結果を8ビットの2進数で表した時の1の数)は奇数
CY : キャリー(結果の桁あふれ)あり

図3-2-21 デバッガによるフラグの表現

さて，CPUの内部では，CMP AX，BXの実行時に次のような減算を行います(図3-2-22)．

AX－BX＝1234H－5678H<0

図3-2-22 データの比較

今，この結果はもちろん負の数になります．そのため符号を表すサイン(Sign)フラグSFがセットされ，NG(Negative：負)を示したのです．さらにこの減算は，そのままではできないため，1234Hは上の桁からの借り(Borrow：ボロー)を生じます．このような桁借りもキャリー(Carry)フラグCFに現れます．

同様に，5ビット目から4ビット目へも桁借りが生じ，この結果補助キャリーがセットされます．

このようなフラグ・レジスタの変化は，マシン語命令の実行によって生じますが，影響されるフラグは個々の命令ごとに決まっています．

ではこのフラグの変化をどのように利用するのでしょうか．このマシン語プログラムは，AXとBXのうち内容の大きい方から小さい方を減算するため，SUB（Subtract）命令を使って，

```
SUB  AX , BX
```

としています．これは，AXからBXを減算し結果をAXに入れる命令です．したがって，もしBXがAXより大きいとき(つまり今の場合)は，AXとBXを入れ換える必要があり，そうでなければ入れ換えは不要です．これをBASICで記述すれば次のようになります．

```
10  IF  AX<BX  THEN  30
20  SWAP  AX , BX
30  AX=AX-BX
```

マシン語でもこれと同様のことを行っているのです．

マシン語プログラムでは，AX<BXの判定はその前のCMP命令でAX−BXを行って，その結果の情報，つまり正か負か，または0がフラグ・レジスタにセットされ，次にフラグの状態によってAXとBXの内容を入れ換える命令を実行するか，またはこれを飛ばすかを行うのです．この命令が10FH番地のJA（Jump if Above）という条件分岐命令です．

このJA命令は，前に行った演算の結果が大きいとき，つまり，今の場合だとAXがBXより大きいときに，命令のあとに書かれたアドレスに分岐を行います．その判定のために，CF(キャリーフラグ）とZF（ゼロフラグ）を調べて，分岐は双方がゼロ（クリア）である場合に実行されます．このとき，同時に命令の最終バイトからジャンプ先のラベルまでの距離がIP(インストラクション・ポインタ)に加えられ，次の命令のあるアドレスを示すようにします．

一方，CF，ZFのどちらか，または両方が1(セット)のときは分岐を行なわず，次の命令を実行します．

このCMP命令の結果は，

ZF=NZ (=1), CF=CY (=1)

なので分岐はせずに次の命令にいきます．

ところでJA命令は，実際のマシン語コードでは，

01110110+<JA命令の次の命令の先頭アドレスからジャンプ先のラベルまでの距離(8ビット)>

となるのですが，プログラムでは，

```
JA  0112
```

のように分岐先のオフセット値を書くだけです．これは，デバッガのアセンブラが自動的に分岐のための距離を計算しているのです．

さて，分岐をしない，つまりAXの値がBXの値より小さいときは，AX−BXの値が負になってしまうので，この値を入れ換えます．これはXCHG (Exchange) 命令で行います．

```
XCHG  AX , BX
```

これによりAXの内容はBXに入り，BXの内容はAXに入ります．この様子を図3-2-23に示します．

```
-T

AX=1234  BX=5678  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=11BC  ES=11BC  SS=11BC  CS=11BC  IP=0111   NV UP DI NG NZ AC PO CY
11BC:0111 93            XCHG    BX,AX      └─前のJA命令による分岐はなかった
-T

AX=5678  BX=1234  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=11BC  ES=11BC  SS=11BC  CS=11BC  IP=0112   NV UP DI NG NZ AC PO CY
11BC:0112 29D8          SUB     AX,BX
-T
        ─SUB命令の結果
AX=4444/ BX=1234  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=11BC  ES=11BC  SS=11BC  CS=11BC  IP=0114   NV UP DI PL NZ NA PE NC
11BC:0114 A30402        MOV     [0204],AX                        DS:0204=0000
-T

AX=4444  BX=1234  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=11BC  ES=11BC  SS=11BC  CS=11BC  IP=0117   NV UP DI PL NZ NA PE NC
11BC:0117 0000          ADD     [BX+SI],AL                       DS:1234=00
-D 200 20F
11BC:0200  34 12  78 56  44 44 00 00-00 00 00 00 00 00 00 00   4.xVDD..........
-               実行結果
```

図3-2-23　実行のトレース(1)

さてこれで減算の準備ができたので，次のSUB命令でこれを行います．結果はAXに格納されるので，これを先程行ったのと同じようにMOV命令で204H番地に格納して完了します．

同様にして，AXの値がBXの値より大きい場合のトレース結果を図3-2-24に示します．このプログラムは"CH3−2"としてセーブしておいたもので，AX=4567H, BX=2345Hとなっています．

```
-R
AX=0000  BX=0000  CX=0017  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=11BC  ES=11BC  SS=11BC  CS=11BC  IP=0100   NV UP DI PL NZ NA PO NC
11BC:0100 B86745        MOV     AX,4567
-T=100,8

AX=4567  BX=0000  CX=0017  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=11BC  ES=11BC  SS=11BC  CS=11BC  IP=0103   NV UP DI PL NZ NA PO NC
11BC:0103 A30002        MOV     [0200],AX                       DS:0200=1234

AX=4567  BX=0000  CX=0017  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=11BC  ES=11BC  SS=11BC  CS=11BC  IP=0106   NV UP DI PL NZ NA PO NC
11BC:0106 BB4523        MOV     BX,2345

AX=4567  BX=2345  CX=0017  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=11BC  ES=11BC  SS=11BC  CS=11BC  IP=0109   NV UP DI PL NZ NA PO NC
11BC:0109 891E0202      MOV     [0202],BX                       DS:0202=5678

AX=4567  BX=2345  CX=0017  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=11BC  ES=11BC  SS=11BC  CS=11BC  IP=010D   NV UP DI PL NZ NA PO NC
11BC:010D 39D8          CMP     AX,BX

AX=4567  BX=2345  CX=0017  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=11BC  ES=11BC  SS=11BC  CS=11BC  IP=010F   NV UP DI PL NZ NA PE NC
11BC:010F 7701          JA      0112

AX=4567  BX=2345  CX=0017  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=11BC  ES=11BC  SS=11BC  CS=11BC  IP=0112   NV UP DI PL NZ NA PE NC
11BC:0112 29D8          SUB     AX,BX

AX=2222  BX=2345  CX=0017  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=11BC  ES=11BC  SS=11BC  CS=11BC  IP=0114   NV UP DI PL NZ NA PE NC
11BC:0114 A30402        MOV     [0204],AX                       DS:0204=4444

AX=2222  BX=2345  CX=0017  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=11BC  ES=11BC  SS=11BC  CS=11BC  IP=0117   NV UP DI PL NZ NA PE NC
11BC:0117 0000          ADD     [BX+SI],AL                      DS:2345=00
-D 200 20F
11BC:0200  67 45 45 23 22 22 00 00-00 00 00 00 00 00 00 00   gEE#''..........
-
```

CMP命令の結果

JA命令により分岐し，XCHG命令は行わない

実行結果

図3-2-24　実行のトレース(2)

この場合は，CMP AX，BX の結果，CF＝NC (0)，ZF＝NZ (0) で分岐条件が成り立つので，AX と BX を交換する必要がなく，XCHG 命令は実行されず，112H 番地へ分岐が行われていることがわかります．

以上は，マシン語プログラムのごく簡単なものではありますが，CPU の動作が基本的にレジスタを中心に行なわれること，フラグという特殊な方法を用いて条件判断を行っていることなどが理解できたと思います．

3-3　DEBUGと機械語モニタの相違点

3-3-1　使用法の相違

PC-9801のN_{88}-BASICには，マシン語プログラム開発用に，機械語モニタモードと呼ばれるモードが用意されています．その機能はほぼMS-DOSのデバッガと同じですが，ここでは機械語モニタを利用してプログラムを入力される方のために，両者の操作上の相違点について説明します．

機械語モニタの起動はBASICのコマンド待ち状態から，MONコマンドで行います．しかし，実際にBASICで使うマシン語プログラムを作成するときには，モニタモードへ移る前に，CLEAR文により，マシン語プログラム・セグメント（マシン語プログラムのための領域）を確保しなくてはなりません．この方法については第6章で説明します．

デバッガでのメモリ・アドレスの指定は，

　　セグメント：オフセット

により行い，メモリ空間のどこでも参照できましたが，機械語モニタでは，まずセグメント値(セグメントベース）を，Cコマンド（チェンジ・セグメント）で指定し，次にDコマンド等でそのセグメントベースに対するオフセットを指定してメモリ参照を行うという2つのステップが必要です．

また，デバッガでは入力する数値は16進数でしたが，モニタではBコマンドにより8進，16進の2通りを切り換えて，入力することができます．

その他にプログラムの入力で注意することは，機械語モニタのAコマンドでは，デバッガで使われるCALL　FAR，JMP　FAR，RETなどの64Kバイトをこえる分岐命令は使用できないことです．

機械語モニタにはこの他，テープへの入出力(W，Rコマンド)，メモリ内容のスクリーンエディット機能（Eコマンド）などがあります．

3-3-2　DEBUGと機械語モニタのコマンドの対応

DEBUGと機械語モニタのコマンドを比較し，まとめたものを表3-3-1に示します．

本書の内容をモニタから入力する際の参考にしてください．

表3-3-1　デバッガと機械語モニタのコマンド一覧

機　能	デバッガコマンド	機械語モニタコマンド
1行のアセンブル	A[〈アドレス〉]	A [〈アドレス〉]
メモリ内容の比較	C 〈レンジ〉〈アドレス〉	———
メモリ内容のダンプ	D[〈アドレス〉[L〈バイト数〉]] D[〈レンジ〉]	D [〈表示開始アドレス〉],[〈表示終了アドレス〉]
データ入力	E 〈アドレス〉[〈データのリスト〉]	S [〈開始アドレス〉]
メモリ内容のフィル	F 〈レンジ〉〈データリスト〉	F 〈開始アドレス〉,〈終了アドレス〉,〈値〉
実行	G[=〈実行開始アドレス〉〈ブレークポイント・アドレス〉……]	G [〈実行開始アドレス〉][,〈ブレークポイント 1〉],[,〈ブレークポイント 2〉]
16進数の和，差を求める	H 〈2バイトデータ〉〈2バイトデータ〉	———
I/Oポートからの入力	I 〈ポート・アドレス(16ビット)〉	I 〈ポート・アドレス(8ビット)〉
ディスクからの読み込み	L [〈アドレス〉[〈ドライブ〉〈レコード〉〈セクタ数〉]]	CTRL + R〈ドライブ#〉[,〈サイド#〉],〈トラック#〉,〈セクタ〉,〈レンジ〉
メモリ中のデータの移動	M 〈レンジ〉〈転送先アドレス〉	M 〈転送元開始アドレス〉,〈転送元終了アドレス〉,〈転送先アドレス〉
名前付け	N 〈ファイル名1〉[〈ファイル名2〉……]	———
I/Oポートへの出力	O 〈ポート・アドレス〉,〈バイトデータ〉	O 〈ポート・アドレス〉,〈バイトデータ〉
デバッガの中止	Q	CTRL + B
レジスタ内容の表示，変更	R[〈レジスタ名〉]	X [〈CPUレジスタ名〉または〈CPUフラグ名〉]
検索	S 〈レンジ〉〈リスト〉	———
実行のトレース	T [=〈アドレス〉][〈ステップ数〉]	———
逆アセンブル	U [〈アドレス〉[L〈バイト数〉]] U [〈レンジ〉]	L [〈逆アセンブル開始アドレス〉][,〈終了アドレス〉]
ディスクへの書き込み	W [〈アドレス〉[〈ドライブ〉〈レコード〉〈セクタ数〉]]	CTRL + W 〈ドライブ#〉[,〈サイド#〉],〈トラック#〉,〈セクタ#〉,〈レンジ〉
セグメントの変更	Rコマンドでセグメント・レジスタを変更する	C [〈セグメントベース・アドレス〉]
プリンタへの出力コントロール	CTRL + P	P
数値表現の変更(8進，16進)	———	B {h (16進) / g (8進)}
メモリのスクリーンエディット	———	E [〈開始アドレス〉]
テープからのリード	———	R [1/2] [〈ファイル名〉]
メモリのテスト	———	TM
テープのベリファイ	———	V [1/2] [〈ファイル名〉]
テープへのライト	———	W [1/2] [〈ファイル名〉],〈開始アドレス〉,〈終了アドレス〉
コマンド使用法の表示	———	CTRL + A または HELP
ディスク内容のダンプ	———	CTRL + D〈ドライブ#〉[,〈サイド#〉],〈トラック#1〉,〈セクタ#1〉,〈トラック#2〉,〈セクタ#2〉

※〈レンジ〉は〈開始アドレス〉,〈終了アドレス〉の意味
デバッガでは，アドレスはセグメント：オフセットで指定する，セグメントは省略可

4章
マシン語プログラミング

第3章でも，簡単なマシン語プログラムが登場しましたが，そこでは，CPU(8086)の基本動作を解説することを目的としたため，マシン語自身の基本的用法については，あえて説明を省略した面がありました．

この章では，前章で使用したデバッガを使って，マシン語の基本用法について解説し，さらにそれらを組み合せていきながら，マシン語プログラムの作り方について学習していこうと思います．そして，その過程を通して，アルゴリズムを記述する際のプログラミング・テクニックの習熟を図っていきます．

4-1　プログラムの基本形

4-1-1　アルゴリズムとフローチャート

マシン語プログラミングが，BASICのプログラミングほど楽なものではないことは，第3章の例でもわかると思います．その理由をまとめると以下のようになります．

○マシン語プログラムは，プログラムエリアを考慮する必要がある
○データの形式や長さに気を使わなくてはならない
○ BASICのように，入力後すぐに実行できない(とくにアセンブラ使用の場合で，デバッガではこの限りでない)
○プログラムの途中を変更するにはその後を書き換えなくてはならない

これら以外にも色々ありますが，とにかく，BASICのスクリーンエディタでプログラミングをするような，対話型の，ともすれば，いきあたりばったりになりがちなプログラム開発はできません．

マシン語プログラミングを効率よく行うには，事前にプログラムの内容，構造を明確にする必要があります．このプログラムの内容，つまり明確で論理的なデータの処理手順のことを，アルゴリズムといいます．

アルゴリズムの記述方法として，古くから知られているものにフローチャートがあり，その記述のために，いくつもの処理に対応した記号が決められています．

フローチャートを用いるのは，次の利点が考えられるからです．

○プログラムの流れを視覚化することによってイメージをつかむことができる

○自分が何のプログラムを書こうとしているのかを認識することができる

○プログラム完成後にもドキュメントとして残るため，他の人々の解釈の手助けになる

これらの利点は，実体がつかみにくく，表現力のとぼしいマシン語プログラムの欠点を補い，プログラム開発の効率向上に大変効果があります．とくに3番目の利点は，億劫になりがちなドキュメント作成も兼ねており，一石二鳥といえるでしょう．

フローチャート記号は，その処理によっていくつかのものが用意されていますが，ここでは，以下の3つを基本とします．

○入口，出口端子

○パス

○条件ブランチ（分岐）

それぞれの記号を図4-1-1に示します．

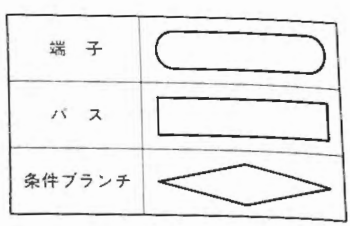


図4-1-1　フローチャート記号

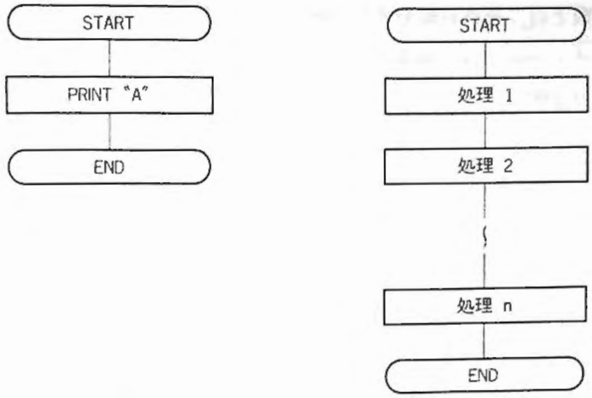
次に，これらの記号の意味と書き方，また，組み合せ方について解説しましょう．

4-1-2　パス（Path）

パス（Path）とは通り道のことですが，ここでは何らかの処理のことをいいます．最も簡単なプログラムの形は，図4-1-2のように1つのパスだけでできたものです．

このように，フローチャートでは，プログラムの流れは，入口端子から始まり，線に沿って下に向かい，パスを通って出口端子で終了するように表されます．この形式ならば，パスがいくら増えてもかまいません．

また1つのパスの長方形が表す処理内容は，単独の命令ではなく，複数の命令の組み合せである場合もあります．



(a)　　　　(b)

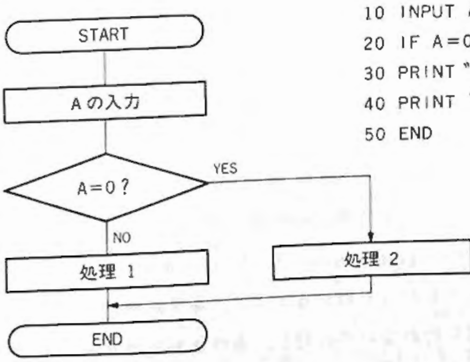
図4-1-2　パス(Path)の例

4-1-3　ブランチ (Branch)

ブランチとは「枝」のことですが，プログラムの実行も木が幹から枝を出すように，いくつかに枝分れ（分岐）します．

一般にこのブランチには条件分岐と無条件分岐があります．

条件分岐とは，ある条件を満たした場合に分岐を行い，そうでなければ次の命令へ行くというもので，フローチャートでは図4-1-3のようにひし形の中に条件を書き，分岐先に入る線には矢印をつけます．条件の結果の "Yes"，"No" の出力方向は，条件の内容の否定をとれば逆になるので，随時，変えてかまいません．この条件分岐は，BASICのIF～THEN～文にあたります．




```
10 INPUT A
20 IF A=0 THEN 40 ELSE 30
30 PRINT "not 0": GOTO 50
40 PRINT "0"
50 END
```

図4-1-3　ブランチ(Branch)の例

一方，無条件分岐とは，条件判断せずに分岐先へ実行を移すもので，BASICのGOTO文にあたります．フローチャートでは，無条件分岐を表す特別な記号はなく，実行の流れを示す線によって表すことになります．

4-1-4 ループ (Loop)

プログラムのアルゴリズムは，以上に述べたパスとブランチの組み合せで記述することができます．ここではその組み合せの例として，プログラムのループ(繰返し)について説明しましょう．

最も単純なループは前出の無条件分岐によって，いくつか前のパスに戻すもので，この結果は，いつまでもそのパスを繰り返す無限ループになります（図4-1-4）．

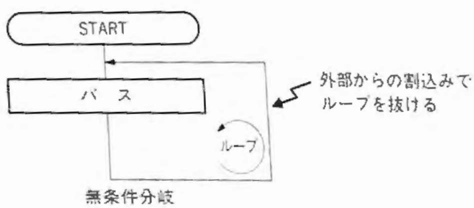


図4-1-4　無限ループの例

また，無条件分岐命令自身に戻るようにすると，表面上は何も行わず，止まっているのと同じことになります．

このプログラムを停止するには，リセットや，割込み（第7章参照）などの外部からの働きかけが必要です．

一方，条件分岐によるループには，条件判断をいつ行うかによって2つの形があります．

そのひとつは，ある条件が成り立っている間は，その処理を繰り返すもので，フローチャートでは図4-1-5のように表されます．このループは，条件判定が処理の前にあるので，条件前判定型ループと呼ぶことにします．構造化言語(PascalやCなどの言語)などでのwhile～do形やBASICでのwhile～wend文がこれにあたります．

もうひとつは，ある条件が成り立つまでその処理を繰り返すもので，フローチャートでは図4-1-6のように表されます．このループは条件判定を処理の後で行うため，条件後判定型ループと呼ぶことにします．構造化言語でのrepeat～until～形がこれにあたります．

この両者の違いは，フローチャートからもわかるように，条件前判定型ループが，最初から条件が成り立たなければ一度も処理を行わないのに対し，条件後判定型ループでは，少なくとも一度はその処理を行うということです．

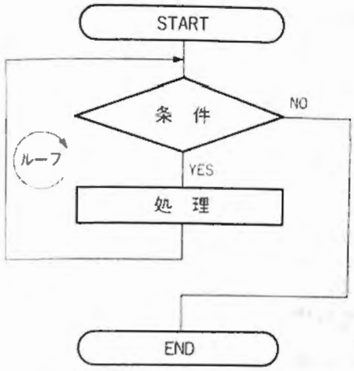


図4-1-5　条件前判定型ループの例

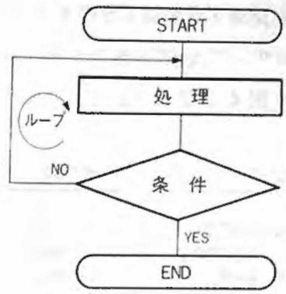


図4-1-6　条件後判定型ループの例

4-2　データを転送する

4-2-1　データの転送とアドレッシングモード

さて，本節からは，いよいよ8086の個々のマシン語命令とそのプログラミングについて解説していきます．

ただし，マシン語命令といっても，16進数で表現されたマシン語コードそのものを扱うのは大変です．そこで，ここでは，16進表現のかわりにアセンブリ言語のニーモニックによる表現を使用します．したがって特に断りがない場合，マシン語プログラムとはアセンブリ言語プログラムのことであると考えてください．

一般に，マシン語プログラムで行う仕事は，データ転送とデータ処理（演算）が大部分を占めます．このデータ転送を行うために，8086で最もよく使われる命令はMOV (Move) 命令と呼ばれるもので，これは，アセンブリ言語では次のようなニーモニック表現をします．

```
MOV　DEST , SORC
```

DESTは，ディスティネーション (Destination) を意味し，SORCはソース (Source) を意味しており，SORCで指定されたデータが，DESTで指定された場所（メモリまたはレジスタ）に転送されます．転送元(SORC)には数値データ，レジスタ名，メモリ・アドレスなどが指定でき，転送先(DEST)には，レジスタ名，メモリ・アドレスが指定できます．第2章でも述べたように，8086では，その指定方法のことをアドレッシングモードと呼んでいます．第2章では，アドレッシングモードの種類について説明しましたが,ここではそれを,実際の例を挙げて解説しましょう．

この章で紹介するサンプル・プログラムは，プログラム中で指定のない限り，デバッガの起動時のセグメント・アドレスを使い，CS，DS，SS，ESの値も同一とすることにします．また，第3章で行ったように，プログラムの入力前に，使用するエリアをFコマンドですべて00Hにして

おくものとし，プログラムはCSのオフセット100H番地から書き込むことにします．

MOV命令のサンプル・プログラムをリスト4-2-1に，フローチャートを図4-2-1に，メモリの使用の様子を図4-2-2にそれぞれ示します．

リスト 4-2-1 サンプル1(データ転送)

```
11BC:0100 B001      MOV   AL,01 ──────── ALに01Hを転送
11BC:0102 A20002    MOV   [0200],AL ── 200H番地にALの値を転送
11BC:0105 B002      MOV   AL,02     ┐
11BC:0107 A20102    MOV   [0201],AL ┘ 02Hを201H番地に転送
11BC:010A B003      MOV   AL,03     ┐
11BC:010C A20202    MOV   [0202],AL ┘ 03Hを202H番地に転送
11BC:010F B004      MOV   AL,04     ┐
11BC:0111 A20302    MOV   [0203],AL ┘ 04Hを203H番地に転送
11BC:0114 B005      MOV   AL,05     ┐
11BC:0116 A20402    MOV   [0204],AL ┘ 05Hを204H番地に転送
11BC:0119 90        NOP ──────────── NOP命令実行の際ブレークポイントが入る
```

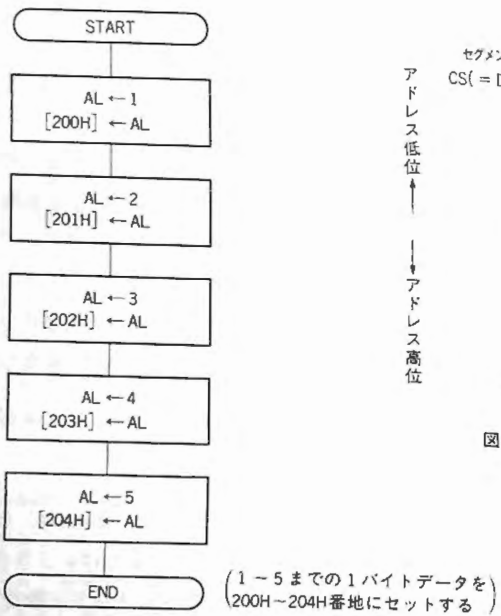


図4-2-1 フローチャート(リスト4-2-1)

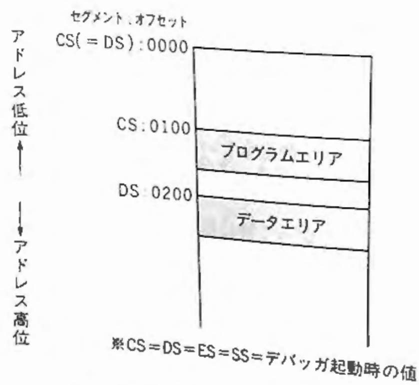


図4-2-2 メモリ使用の様子(マップ)

このプログラムは，1から5の1バイトデータを，MOV命令を使ってメモリのオフセット200H番地からに格納するプログラムで，2種類のアドレッシングモードを使用しています．その1つは，1行目で代表されるイミディエイトモードのMOV命令です．これは，ALレジスタに1Hという1バイトの値を転送することを意味しており，イミディエイト(直接)モードとはこのように，SORC

部にある直接の値を，DEST 部に示された場所（アドレスまたはレジスタ）に代入する命令モードを意味しています．MOV におけるイミディエイトモードの一般表記は，

MOV　DEST，バイトまたはワードデータ

となります．

もう1つのモードは2行目に代表されるダイレクトモードの MOV 命令です．これは，SORC 部にある AL の値を，DEST 部で示されたアドレスに書き込む命令モードを意味します．ここで，200 番地というアドレスは数値としての 200 と区別するために，角カッコでくくって表します．

またこのときのオフセット･アドレス 200H は，何も指定しなければ DS の値をセグメントベースとすることになっています．プログラムが置かれるのは CS で指定したセグメントベースなので，CS と DS の値を違えることによって，まったく別のセグメントにアクセスすることができるようになっています．ただしここでは，CS，DS，ES，SS が同じなので区別はつきません．

ダイレクトモードの一般表記は，

MOV　DEST，[アドレス]

または，

MOV　[アドレス]，SORC

となります．このように，アドレッシングモードは，DEST 部，SORC 部それぞれについて設定されます．ただし DEST 部がイミディエイト（直値）モードになることはありません．

このプログラムを入力して，

－G=100，119⏎

で実行したのち，D コマンドでメモリを見てみると，200H から，1～5 の値が格納されているのがわかります（図 4-2-3）．

```
-g=100,119
           ブレークポイント
AX=0005  BX=0000  CX=001A  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=11BC  ES=11BC  SS=11BC  CS=11BC  IP=0119   NV UP DI PL NZ NA PO NC
11BC:0119 90                 NOP
-d 200          転送されたデータ
11BC:0200  01 02 03 04 05 00 00 00-00 00 00 00 00 00 00 00   ................
11BC:0210  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
11BC:0220  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
11BC:0230  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
11BC:0240  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
11BC:0250  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
11BC:0260  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
11BC:0270  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
```

図 4-2-3　リスト 4-2-1 の実行結果

4-2-2 メモリ間のデータ転送

MOV 命令を使って,図 4-2-5 のように,あるメモリ領域のデータを別の領域にそっくり転送するプログラムを作ってみます（リスト 4-2-2, 図 4-2-4, 4-2-5).

リスト 4-2-2 サンプル2(データ転送)

```
11BC:0100 A00002         MOV     AL,[0200]———ALに200H番地の内容を転送
11BC:0103 A28002         MOV     [0280],AL———ALの値を280H番地に転送
11BC:0106 A00102         MOV     AL,[0201]  ┐
11BC:0109 A28102         MOV     [0281],AL  ┘201H番地の内容を281H番地に転送
11BC:010C A00202         MOV     AL,[0202]  ┐
11BC:010F A28202         MOV     [0282],AL  ┘202H番地の内容を282H番地に転送
11BC:0112 A00302         MOV     AL,[0203]  ┐
11BC:0115 A28302         MOV     [0283],AL  ┘203H番地の内容を283H番地に転送
11BC:0118 A00402         MOV     AL,[0204]  ┐
11BC:011B A28402         MOV     [0284],AL  ┘204H番地の内容を284H番地に転送
11BC:011E 90             NOP———————————————実行の際ブレークポイントを置く
```

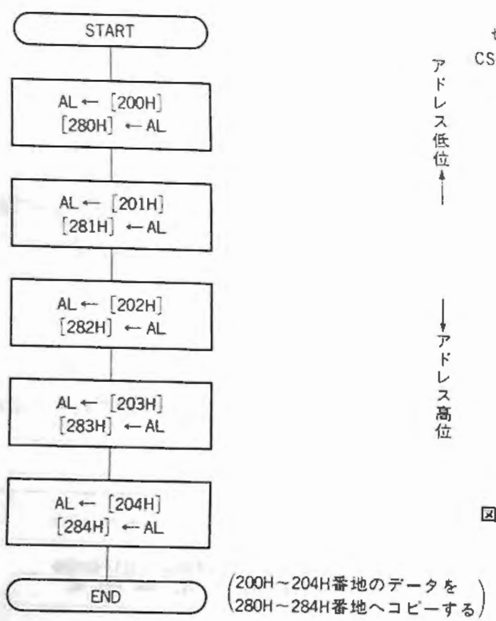


図 4-2-4 フローチャート(リスト4-2-2)

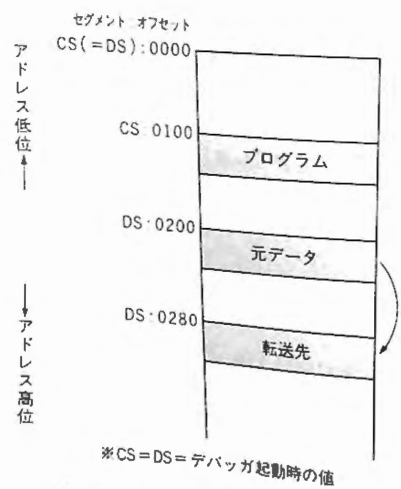


図 4-2-5 MOV 命令を使ったブロック転送

このプログラムで使用しているアドレッシングモードはダイレクトモードのみです。基本的なデータ転送は 1 行目と 2 行目で代表されるように，まず，先のリスト 4-2-1 の実行によって作成した，1~5 のデータのある 200H 番地からのメモリの内容を順次 AL に転送し，次に AL の内容を転送先のアドレス 280H 番地からにダイレクトモードで順次書き込んでいます。これをデータ領

域のすべてに対して行っています．

デバッガで，

　　−G=100，11E[↵]

として実行したのち，Dコマンドでメモリをダンプすると，データが転送されているのがわかります（図4-2-6）．

```
-g=100,11e

AX=0005  BX=0000  CX=001F  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=11BC  ES=11BC  SS=11BC  CS=11BC  IP=011E   NV UP DI PL NZ NA PO NC
11BC:011E 90            NOP
-d200,2ff        もとのデータ
11BC:0200  01 02 03 04 05 00 00 00-00 00 00 00 00 00 00 00   ................
11BC:0210  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
11BC:0220  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
11BC:0230  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
11BC:0240  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
11BC:0250  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
11BC:0260  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
11BC:0270  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
11BC:0280  01 02 03 04 05 00 00 00-00 00 00 00 00 00 00 00   ................
11BC:0290  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
11BC:02A0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
11BC:02B0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
11BC:02C0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
11BC:02D0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
11BC:02E0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
11BC:02F0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
-                  └─転送されたデータ
```

図4-2-6　リスト4-2-2の実行結果

4-2-3　ループを構成する

リスト4-2-2のデータ領域転送プログラムでは，1回のデータ転送のたびにMOV命令を2回ずつ書かなくてはなりませんでした．もしデータ数が数十バイトにもなった場合には大変な手間となり，実用的なプログラムとはいえません．

そこで，基本となる2つのMOV命令をループの中で繰り返すことによって，データ転送を行うプログラムを作ってみましょう．プログラムをリスト4-2-3に示します（図4-2-7）．

リスト 4-2-3　サンプル3（ループによる転送）

```
11BC:0100 BE0400        MOV    SI,0004
11BC:0103 8A840002      MOV    AL,[SI+0200]—DS:[SI+200H]番地の内容をALに転送
11BC:0107 88840002      MOV    [SI+0280],AL—ALの値をDS:[SI+280H]番地に転送
11BC:010B 4E            DEC    SI ————————SIを1へらす
11BC:010C 79F5          JNS    0103————————正ならば103H番地へ分岐
11BC:010E 90            NOP
```

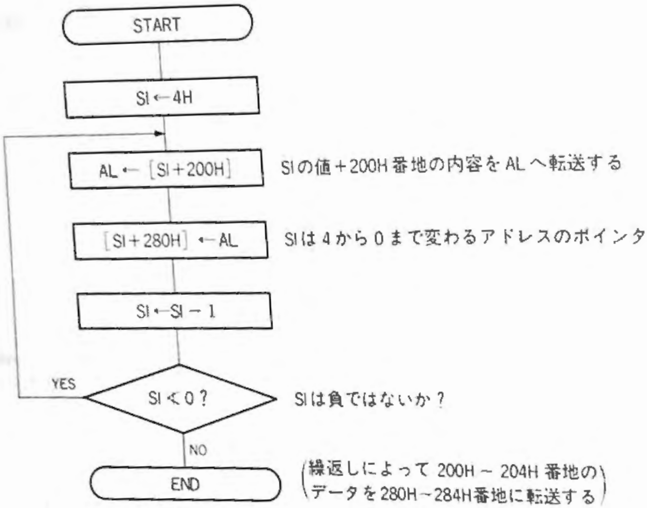


図4-2-7　フローチャート(リスト4-2-3)

このプログラムのメモリ使用状態は，図4-2-5と同じで，200～204H番地のデータ1～5を280～284H番地へ転送します．プログラムの流れは，図4-2-7に示すように，ソースインデックス・レジスタSIの値を4から0まで1ずつ減らしながら，[SIの内容+200H]番地の内容を［SIの内容+280H]番地へと転送するものです．

アドレスを示すときに，このSIのような指標（インデックス）を用いて，間接的に示すアドレッシングを，インダイレクト（間接）アドレッシングモードと呼びます．

インダイレクトモードで使用するインデックス・レジスタにはSIの他にDIがありますが，この2本の16ビット・レジスタは，ここでの場合のように，ベースになるある値に対する変数オフセットとして使用しアドレス指定を行います．

インデックス・レジスタを使用したインダイレクトモードの表記は，

MOV　DEST，[I-Reg+d]

または，

MOV　[I-Reg+d]，SORC

(I-Regは，SIまたはDI．dはディスプレイスメントと呼ばれる，8ビットまたは16ビットで表された，ベースとなるアドレスを表す数値)

となります．

このときDESTには，レジスタのみ，SORCには，レジスタとイミディエイトデータが指定できます．

プログラムのループ中で，インデックス・レジスタSIの値を1ずつ減じているのが，DEC(Decrement

by 1）命令です．DEC 命令はオペランドとして，ディスティネーションのみを伴います．

続いて書かれた，JNS（Jump-on-Not-Sign）は，条件分岐命令のひとつで，演算結果の正負を示すサインフラグ SF が 0(正)の場合に，JNS に続いて書かれたアドレスに分岐するものです．

この場合は，最初に 4 だった SI の値が前述の DEC 命令で減じられながら，この JNS 命令によってループを回り，5 回目の DEC 命令で SI が負になった時，SF が 1 になり，JNS 命令をぬけて，実行を終了するのです．

```
-G=100，10E⏎
```

とした時の実行結果を，図 4-2-8 に示します．

```
-g=100,10e

AX=0001  BX=0000  CX=000F  DX=0000  SP=FFEE  BP=0000  SI=FFFF  DI=0000
DS=11BC  ES=11BC  SS=11BC  CS=11BC  IP=010E   NV UP DI NG NZ AC PE NC
11BC:010E 90            NOP
-d200,20f
11BC:0200  01 02 03 04 05 00 00 00-00 00 00 00 00 00 00 00   ................
-d280,28f      もとのデータ
11BC:0280  01 02 03 04 05 00 00 00-00 00 00 00 00 00 00 00   ................
-              転送されたデータ
```

図 4-2-8　リスト 4-2-3の実行結果

リスト 4-2-3 のループによるデータ転送は別のインダイレクトモードの MOV 命令で書くこともできます．そのいくつかを紹介しましょう（リスト 4-2-4，図 4-2-9）．

リスト 4-2-4　サンプル 4（ループによる転送）

```
11BC:0100 BB0002        MOV     BX,0200
11BC:0103 BE0400        MOV     SI,0004
11BC:0106 8A00          MOV     AL,[BX+SI]——DS：[BX+SI]番地の内容をALに転送
11BC:0108 88808000      MOV     [BX+SI+0080],AL——ALの内容をDS:[BX+SI+80H]
                                                     番地に転送
11BC:010C 4E            DEC     SI ————SIを1へらす
11BC:010D 79F7          JNS     0106——負でなければ106H番地へ分岐
11BC:010F 90            NOP
```

リスト 4-2-4 のプログラムは，リスト 4-2-3 と同じく MOV 命令のインダイレクトモードを使用していますが，その中でもベース・レジスタ BX と SI を使ったものです．

リスト 4-2-3 では，データの転送元および転送先のアドレスが固定されたものであるのに対し，ここでは BX と SI の内容の和がアドレスを示すので，1 行目で BX にセットする値を変えるだけで，セグメント内にある距離 80H 隔てたデータ領域間でのデータを転送することができます．

BX と SI（DI）およびディスプレイスメントを使った MOV 命令のインダイレクトモードの表記は，

MOV　DEST，[BX+I-Reg+d16 or d8]
MOV　[BX+I-Reg+d16 or d8]，SORC
(I-Reg は SI または DI，d16 は 16 ビットの数値，d8 は 8 ビットの数値)

です．DEST，SORC 部どちらにでもこのモードを使うことができますが，両方同時には使用できません．

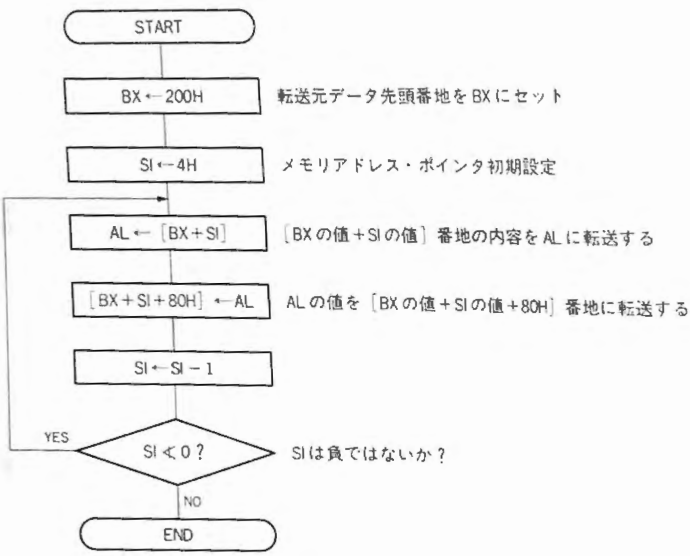


図4-2-9　フローチャート(リスト4-2-4)

インダイレクトモードには，もうひとつ，ベースページ・レジスタ BP を使用したものも可能です．これを使用すると，さらにプログラムの自由度，汎用性は拡張します（リスト 4-2-5，図 4-2-10）．

リスト 4-2-5　サンプル5（ループによる転送）

```
11BC:0100 BB0002        MOV     BX,0200
11BC:0103 BD8002        MOV     BP,0280
11BC:0106 BE0400        MOV     SI,0004
11BC:0109 8A00          MOV     AL,[BX+SI]
11BC:010B 3E            DS:  ──────── 次の命令のオペランドは，セグメントをDSとする
11BC:010C 8802          MOV     [BP+SI],AL── ALの値をDSの[BP+SI]番地に転送する
11BC:010E 4E            DEC     SI
11BC:010F 79F8          JNS     0109
11BC:0111 90            NOP
```

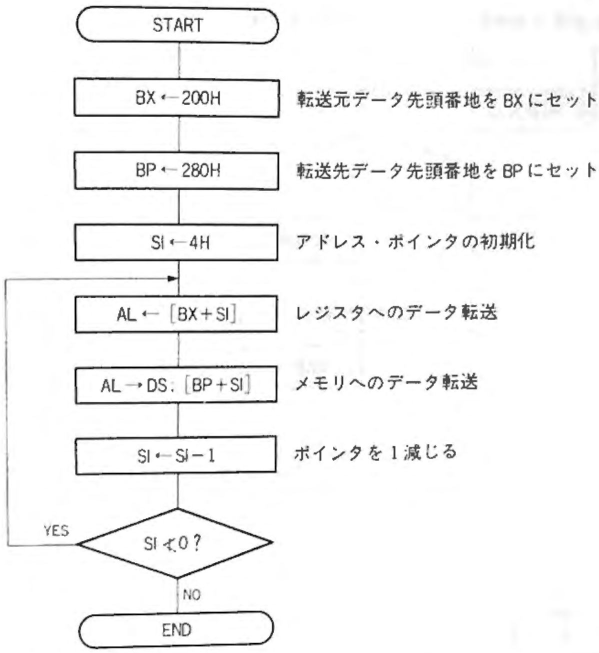


図4-2-10　フローチャート(リスト4-2-5)

リスト4-2-5では，BPに転送先データ領域のベースアドレスをセットしているため，この値をプログラム中で自由に変えれば，同じセグメント内なら，どこにでもデータを転送できます．ここで，

```
DS:
MOV  [BP+SI], AL
```

のような表現が初めて出てきました．これがセグメント・オーバーライド・プレフィックスというものです．BPにこれが必要な理由は，アドレス参照命令において大部分のレジスタがDSの値をデフォルトのセグメントベースとするのに対し，BPを使ったインダイレクトモードでは，SSの値がセグメントベースになるからです．したがってDSをセグメントベースにしたい時は，そのオペランドの前にDS：をつけて，指定をする必要があるのです．これは，CS, SS, ESなどにもいえることで，例えばCSとDSが異なっているときに，プログラムがCS中のデータ領域のデータを必要とする場合には，

```
CS:
MOV  AX, [BX+SI+2]
```

のように指定します．

図4-2-11にリスト4-2-3, 4, 5それぞれのプログラムの機能の差をまとめておきました．

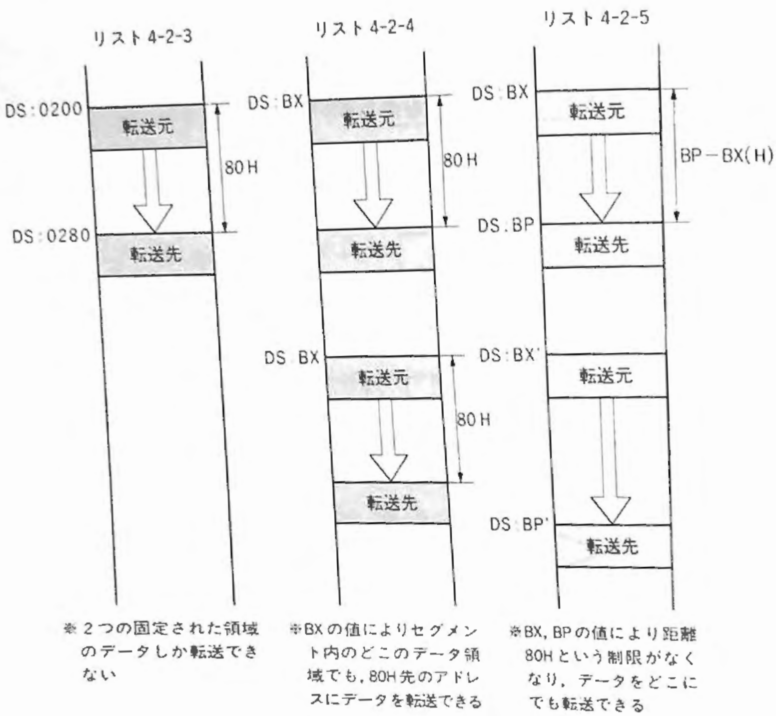


図4-2-11　アドレッシングモードの違いによるプログラムの機能の差

ここでインダイレクトモードのアドレス指定の形をもう一度まとめておくことにしましょう．この表さえ頭に入れておけば，インダイレクトモードは十分使いこなせるでしょう（表4-2-1）．

表4-2-1　インダイレクトモードで使用されるレジスタと表現

レジスタのみ	レジスタ＋d8	レジスタ＋d16
[BX＋SI]	[BX＋SI＋d8]	[BX＋SI＋d16]
[BX＋DI]	[BX＋DI＋d8]	[BX＋DI＋d16]
[BP＋SI]	[BP＋SI＋d8]	[BP＋SI＋d16]
[BP＋DI]	[BP＋DI＋d8]	[BP＋DI＋d16]
[SI]	[SI]＋d8	[SI]＋d16
[DI]	[DI]＋d8	[DI]＋d16
[BX]	[BX]＋d8	[BX]＋d16
	[BP]＋d8	[BP]＋d16

※BP単独のインダイレクト([BP])に対応するマシン語は存在しないが，指定した場合は[BP＋0]となる
d8，d16は8ビットまたは16ビットの直値

データ転送プログラムでデータ領域を指定するのに，セグメント・レジスタを使う方法もあります．これはリスト4-2-6のプログラムのように，DS，ESの2つのセグメント・レジスタに，それぞれ転送元，転送先の領域のベース・アドレスをセットしてインダイレクトモードでアドレッシングする方法で，セグメント間のデータ転送が可能になります．

セグメント・レジスタへの値の代入はイミディエイトモードでは行えず，一度AXなどに転送してから行います．

リスト 4-2-6　サンプル6(セグメント・レジスタを使った転送)

```
11BC:0100 B80010        MOV     AX,1000 ┐DSに1000Hを転送
11BC:0103 8ED8          MOV     DS,AX   ┘
11BC:0105 B80020        MOV     AX,2000 ┐ESに2000Hを転送
11BC:0108 8EC0          MOV     ES,AX   ┘
11BC:010A BE00F0        MOV     SI,F000
11BC:010D 3E            DS:             ┐DSの[SI]番地の内容をALに転送
11BC:010E 8A04          MOV     AL,[SI] ┘
11BC:0110 26            ES:             ┐ESの[SI]番地にALの内容を転送
11BC:0111 8804          MOV     [SI],AL ┘
11BC:0113 4E            DEC     SI
11BC:0114 75F7          JNZ     0100 ――― F000H(61440)回繰り返す
11BC:0116 90            NOP
```

リスト中のエキストラセグメント・レジスタESは，他のセグメント・レジスタと異なり，後述のストリング命令で使用される他は決まった用法はなく，この例のように別のセグメントベースを指定したいときなどの汎用的な用法が可能です．

4-2-4　2バイトデータの転送

これまでのデータ転送プログラムはALを使った1バイトのデータ転送でしたが，これには他の汎用8ビット・レジスタを使用してもかまいません．ただしBXレジスタをアドレッシングで使用するときにはBH，BLは使用できません．

ところで，8086には，AX，BXなど16ビット（2バイト）長のレジスタがあるので，前述のプログラムを少し変更するだけで，2バイト（ワード）データの転送が行えます．

リスト4-2-7はこのプログラムの例です（図4-2-12）．

ワードデータ転送では，データは2バイトずつ転送されるので，アクセスされるアドレスの値は2つずつ変化させなくてはなりません．このプログラムはリスト4-2-5と同じアドレッシングモードを使い，5ワード(10バイト)のデータを転送するもので，データ転送用のレジスタにAXを使用しています．SIの初期値は，(転送ワード数−1)の2倍とし，1回のデータ転送ごとにSIの値を2度のDEC命令で，2つずつ減じています．

リスト 4-2-7 サンプル 7(2バイトデータの転送)

```
11BC:0100 BB0002        MOV     BX,0200
11BC:0103 BD8002        MOV     BP,0280
11BC:0106 BE0800        MOV     SI,0008        DSの[BX+SI]番地の内容を下位
11BC:0109 8B00          MOV     AX,[BX+SI] ― [BX+SI]+1番地の内容を上位として
11BC:010B 3E            DS:                    AXに転送
11BC:010C 8902          MOV     [BP+SI],AX ― AXの値を,DSの[BP+SI]番地に下位
11BC:010E 4E            DEC     SI             [BP+SI]+1番地に上位として転送
11BC:010F 4E            DEC     SI
11BC:0110 79F7          JNS     0109
11BC:0112 90            NOP
```

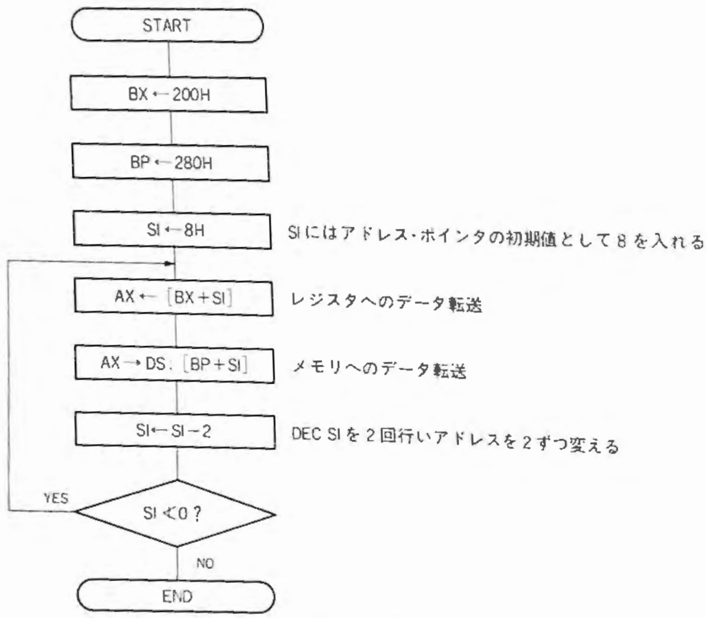


図4-2-12 フローチャート(リスト4-2-7)

−G=100, 112⏎

として実行した結果を図4-2-13に示します.

```
-g=100,112

AX=0100  BX=0200  CX=0013  DX=0000  SP=FFEE  BP=0280  SI=FFFE  DI=0000
DS=11BC  ES=11BC  SS=11BC  CS=11BC  IP=0112   NV UP DI NG NZ NA PO NC
11BC:0112 90            NOP
-d200,20f
11BC:0200  00 01 00 02 00 03 00 04-00 05 00 06 00 00 00 00   ................
                        もとのデータ
-d280,28f
11BC:0280  00 01 00 02 00 03 00 04-00 05 00 00 00 00 00 00   ................
-                      転送されたデータ
```

図4-2-13 リスト4-2-7の実行結果

これまでのサンプル・プログラムでは，ループにはSIレジスタをカウンタとして使い，その内容が負になったかどうかを条件分岐命令で判断してループを行っていましたが，8086にはCXを専用カウンタとして繰返しを行う，専用の命令があります．これを使ってワードデータのブロック転送を行ってみましょう．

この命令はLOOP命令といい，リスト4-2-8のようにCXにループの回数をセットし，繰り返す命令の最後にLOOP命令を書き，ループの先頭アドレスを指定します．すると1回のループごとにCXの値がデクリメントされ，0になると繰返しを抜けます（図4-2-14）．

リスト 4-2-8 サンプル 8（LOOP命令を使った転送）

```
11BC:0100 BB0002        MOV     BX,0200
11BC:0103 BD8002        MOV     BP,0280
11BC:0106 B90500        MOV     CX,0005 ———— カウンタであるCXに5をセット
11BC:0109 89CE          MOV     SI,CX
11BC:010B 4E            DEC     SI
11BC:010C 01F6          ADD     SI,SI ———— SIの値を2倍する
11BC:010E 8B00          MOV     AX,[BX+SI]
11BC:0110 3E            DS:
11BC:0111 8902          MOV     [BP+SI],AX
11BC:0113 E2F4          LOOP    0109 ———— CXの値を1ずつ減じながら0になるまで
                                          109H番地から113H番地までを繰り返す
11BC:0115 90            NOP
```

このプログラムでは，CXをループのカウンタとし，SIはアドレスのポインタとしてだけ使っています．CXには転送語数そのものの5をセットして，ループのたびに，データ領域の先頭からのオフセットが2ずつ変化するので，CXをSIにコピーしてからSIの値から1をひき（DEC SI），ADD（Addition）命令で自分自身を加え，2倍の値を求めています．ADDは加算命令で，これについては後述します．この結果，SIは転送するデータのアドレスを示すオフセット値となります．そして，転送後，LOOP命令でCXがデクリメントされ，CX＝0となるまで109H番地へジャンプします．これで，SIは，8, 6, 4, 2, 0と変化することになります．

LOOP命令は，カウンタとしているCXの内容が0になればループから抜ける命令ですが，ループ命令には，この他に，CXが0でなくてもZF（ゼロフラグ）が0のときにループから抜けるLOOPZ（LOOPE）と，CXが0でなくてもZFが1のときにループから抜けるLOOPNZ（LOOPNE）があります．

以上が，データ転送を行う基本プログラミングです．最後に表4-2-2にMOV命令をまとめておきましょう．

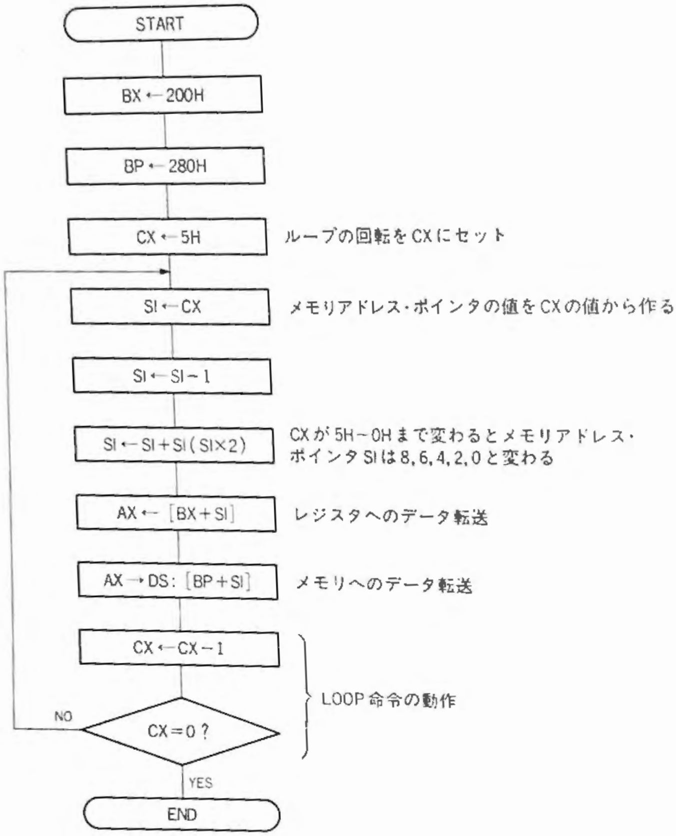


図4-2-14 フローチャート(リスト4-2-8)

表4-2-2 MOV命令のまとめ

DEST＼SORC	レジスタ	セグメント・レジスタ	メモリ	イミディエイトデータ
レジスタ	MOV Reg, Reg	MOV Reg, S·Reg	MOV Reg, mem	MOV Reg, data
セグメント・レジスタ（CS以外）	MOV S·Reg, Reg	————	MOV S·Reg, mem	————
メモリ	MOV mem, Reg	MOV mem, S·Reg	MOVS* mem, mem	MOV mem, data

※転送データのサイズはSORC, DESTともに同じであること
セグメント・レジスタにはイミディエイトデータは転送できない
セグメント・レジスタ間の転送はできない
S·Regはセグメント・レジスタ，Regはそれ以外のレジスタをさす

＊メモリ−メモリ間のデータ転送は後述のMOVS命令で行う

4-2-5 メモリ間転送命令を使う

前項までのデータ転送プログラムでは，MOV 命令を使用して解説してきました．しかし MOV 命令ではメモリ－メモリ間転送が行えず，一度 AX などのレジスタ上にデータを転送してから，それをメモリに転送しなければならないので，1 回のデータ転送に 2 つの MOV 命令が必要でした．しかし 8086 には，これを 1 命令で実行できる MOVS（Move-String）命令があります．

この命令の動作は，ソース・インデックス SI をオフセット・アドレスとするデータ・セグメント DS 内のメモリの内容（ソース・ストリング）を，ディスティネーション・インデックス DI をオフセットとするエクストラ・セグメント ES 内のメモリに転送します．

このときディレクション・フラグ DF が 1（セット）ならば SI，DI はともにデクリメントされ，0（クリア）ならば SI，DI ともにインクリメントされます．つまり，DF は転送をメモリアドレスの小さくなる方向に行うか，大きくなる方向に行うかを決定するのです．さらにこのとき，データがバイトならば SI，DI は自動的に 1 ずつ増減し，ワードなら 2 ずつ増減します．

この MOVS 命令のように，SI，DI という 2 つのレジスタをポインタとして，それを自動的にインクリメント（デクリメント）しながらデータの処理を行う命令をストリング命令といい，8086 の特徴ある命令のひとつです．

この MOVS 命令には，バイトデータ転送用の MOVSB（Move-String-Byte）とワードデータ転送用の MOVSW（Move-String-Word）があります．

この命令の使用例をリスト 4-2-9 に示します．

リスト 4-2-9 サンプル 9（MOVS命令による転送）

```
11BC:0100 FC          CLD ────────── ストリング命令のポインタSI,DIを増加方
11BC:0101 BE0002      MOV    SI,0200   向(インクリメント)にセット
11BC:0104 BF8002      MOV    DI,0280
11BC:0107 B91000      MOV    CX,0010── カウンタ=16
11BC:010A F3          REPZ            ┐SI,DIを1ずつ増加させながら
11BC:010B A4          MOVSB           ┘DS：SI番地の内容のES：DI番地への転
11BC:010C 90          NOP              送を10H(16)回転繰り返す
```

リスト 4-2-9 は 200H 番地から始まるバイトデータを 10H つまり 16 バイト分，280H 番地からの 16 バイトに転送する例で，SI に 200H，DI に 280H を入れ，CLD（Clear-Direction）命令で DF＝0（クリア）とし，CX に転送バイト数を設定して REP 命令で MOVSB 命令を実行します．CLD 命令は DF をクリアする命令で，アドレスがインクリメントされます．

ここで気をつけることは，DI のベース・セグメントが ES であることです．今の場合，DS＝ES なので，問題はありませんが，一般には ES の設定を必要とします．

REP（Repeat-String-op.）命令は，LOOP 命令のように CX をカウンタとするストリング命令用のループ命令のひとつで，CX にループ数を入れて REP 命令の後に，MOVS 命令を行うことで，CX の値だけ MOVS 命令が実行されます．デバッガの T コマンドで確かめれば，MOVS 命令の

繰返しとSI，DIの値の変化がわかるはずです．

リピート命令には，REPと同様の働きをするREPZ（REPE）と，ZFがセットされるとループから抜けるREPNZ（REPNE）があります．

リスト4-2-10はワードデータの転送の場合で，MOVS命令はMOVSWとし，今度はSTD（Set-Direction）命令でDFをセット（1）しアドレスをデクリメントします．またESの値をDSと別にしてデータ転送を行っています．

なお，これらのサンプルを実行するときには，まずEコマンドで転送データをセットしてから行って下さい（図4-2-15）．

リスト 4-2-10 サンプル10（MOVS命令による転送）

```
11BC:0100 FD            STD ──────────── ポインタSI,DIを減少方向(デクリメント)にセット
11BC:0101 8CD8          MOV     AX,DS    ┐
11BC:0103 054000        ADD     AX,0040  │ DS+40HをESとする
11BC:0106 8EC0          MOV     ES,AX    ┘
11BC:0108 BEFE02        MOV     SI,02FE
11BC:010B BFFE00        MOV     DI,00FE
11BC:010E B98000        MOV     CX,0080 ──── カウンタ=80H(128)
11BC:0111 F3            REPZ             ┐ SI,DIの値を2ずつ減少させながら，CXを
11BC:0112 A5            MOVSW            ┘ 1ずつ減じて0になるまでDS:[SI]→ES:[DI]を繰り返す
11BC:0113 90            NOP
```

```
-g=100,113

AX=11FC  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=01FE  DI=FFFE
DS=11BC  ES=11FC  SS=11BC  CS=11BC  IP=0113   NV DN DI PL NZ NA PE NC
11BC:0113 90            NOP
-d200,2ff        └─ESの値が変わっている
11BC:0200  4C 2C 1D 09 49 0A B4 0E-E5 07 80 FD 3E 0A B4 0E   L...I.4.e..>>.4.
11BC:0210  2B 08 80 FD D0 59 60 00-06 59 60 00 2B 08 80 FD   +..)PY`..Y`.+..)
11BC:0220  5D 19 80 FD 74 0D 80 FD-2C 08 80 FD 2C 08 80 FD   ]..)t..),..),..)
11BC:0230  52 4B 60 00 2C 08 80 FD-29 48 60 00 2C 08 80 FD   RK`.,..))H`.,..)
11BC:0240  2C 08 80 FD 2C 08 80 FD-2C 08 80 FD 58 03 00 D7   ,..),..),..)X..W
11BC:0250  2C 08 80 FD 2C 08 80 FD-2C 08 80 FD 2C 08 80 FD   ,..),..),..),..)
11BC:0260  B2 09 80 FD EF 49 60 00-42 46 60 00 84 19 80 FD   2..)oI`.BF`....)
11BC:0270  04 18 80 FD 2B 08 80 FD-EC 27 80 FD EC 27 80 FD   ...)+..)l'.)l'.)
11BC:0280  CF 05 1D 09 D3 05 1D 09-4A 02 B4 0E 78 02 B4 0E   O...S...J.4.x.4.
11BC:0290  BA 04 E3 0D EF 07 1D 09-29 08 1D 09 07 35 1D 09   :.c.o...)....5..
11BC:02A0  E3 05 1D 09 66 8A 60 00-00 00 00 00 00 00 00 00   c...f.`.........
11BC:02B0  00 00 00 00 00 00 00 00-5A 02 E3 0D 00 00 00 00   ........Z.c.....
11BC:02C0  EA E4 05 1D 09 8A 60 00-C5 8A 60 00 C5 8A 60 00   jd....`.E.`.E.`.
11BC:02D0  C5 8A 60 00 C5 8A 60 00-C5 8A 60 00 C5 8A 60 00   E.`.E.`.E.`.E.`.
11BC:02E0  C5 8A 60 00 C5 8A 60 00-C5 8A 60 00 C5 8A 60 00   E.`.E.`.E.`.E.`.
11BC:02F0  C5 8A 60 00 C5 8A 60 00-C5 8A 60 00 C5 8A 60 00   E.`.E.`.E.`.E.`.
-des:0,ff  ──ESを指定                もとのデータ
11FC:0000  4C 2C 1D 09 49 0A B4 0E-E5 07 80 FD 3E 0A B4 0E   L...I.4.e..>>.4.
11FC:0010  2B 08 80 FD D0 59 60 00-06 59 60 00 2B 08 80 FD   +..)PY`..Y`.+..)
11FC:0020  5D 19 80 FD 74 0D 80 FD-2C 08 80 FD 2C 08 80 FD   ]..)t..),..),..)
11FC:0030  52 4B 60 00 2C 08 80 FD-29 48 60 00 2C 08 80 FD   RK`.,..))H`.,..)
11FC:0040  2C 08 80 FD 2C 08 80 FD-2C 08 80 FD 58 03 00 D7   ,..),..),..)X..W
11FC:0050  2C 08 80 FD 2C 08 80 FD-2C 08 80 FD 2C 08 80 FD   ,..),..),..),..)
11FC:0060  B2 09 80 FD EF 49 60 00-42 46 60 00 84 19 80 FD   2..)oI`.BF`....)
11FC:0070  04 18 80 FD 2B 08 80 FD-EC 27 80 FD EC 27 80 FD   ...)+..)l'.)l'.)
11FC:0080  CF 05 1D 09 D3 05 1D 09-4A 02 B4 0E 78 02 84 0E   D...S...J.4.x.4.
11FC:0090  BA 04 E3 0D EF 07 1D 09-29 08 1D 09 07 35 1D 09   :.c.o...)....5..
```

```
11FC:00A0  E3 05 1D 09 66 8A 60 00-00 00 00 00 00 00 00 00
11FC:00B0  00 00 00 00 00 00 00 00-5A 02 E3 0D 00 00 00 00   c...f.`.........
11FC:00C0  EA E4 05 1D 09 8A 60 00-C5 8A 60 00 C5 8A 60 00   ........Z.c.....
11FC:00D0  C5 8A 60 00 C5 8A 60 00-C5 8A 60 00 C5 8A 60 00   jd....`.E.`.E.`.
11FC:00E0  C5 8A 60 00 C5 8A 60 00-C5 8A 60 00 C5 8A 60 00   E.`.E.`.E.`.E.`.
11FC:00F0  C5 8A 60 00 C5 8A 60 00-C5 8A 60 00 C5 8A 60 00   E.`.E.`.E.`.E.`.
-                        転送されたデータ                       E.`.E.`.E.`.E.`.
```

図4-2-15　リスト4-2-10の実行結果

4-3　データの比較と分岐

4-3-1　データの比較と分岐

ここでは，データの大小を比較し処理を行う方法について考えていきます．

ところでマシン語の1命令で比較などの処理が可能なデータには，1バイトデータと2バイトデータがあります．そして，マシン語命令の中には，この処理の結果を符号なしの値として処理する命令と，0を中心とした正負の符号つきの値とみなして処理する命令とがあり，一見似たような処理をする命令でも，数値の取り扱い方の違いにより異なる処理結果を起こす場合があるので注意が必要です．

リスト4-3-1に示したプログラムは，256バイトのメモリ領域中の内容を符号なしの1バイトデータ（0～FFH）として，ある値と比較して大小を調べ，それ以上のもののデータ群と，それ未満のもののデータ群に分けるプログラムです（リスト4-3-1，図4-3-1）．

リスト 4-3-1　大小分類プログラム

```
11BC:0100 8CC8      MOV   AX,CS           ┐
11BC:0102 050002    ADD   AX,0200         │CS+200HをDSに転送
11BC:0105 8ED8      MOV   DS,AX           ┘
11BC:0107 B480      MOV   AH,80 ───────── 比較する値80H
11BC:0109 BB0000    MOV   BX,0000
11BC:010C BE0000    MOV   SI,0000
11BC:010F BF0000    MOV   DI,0000
11BC:0112 81FB0001  CMP   BX,0100 ──────── 比較されるデータの数100H(256)個
11BC:0116 7415      JZ    012D
11BC:0118 8A07      MOV   AL,[BX]
11BC:011A 43        INC   BX
11BC:011B 3AC4      CMP   AL,AH ────────── AH(80H)とAL(データ)を比較
11BC:011D 7207      JB    0126
11BC:011F 88850002  MOV   [DI+0200],AL ┐ 80H以上のデータをDS:[DI+200H]に格納
11BC:0123 47        INC   DI           ┘
11BC:0124 EB05      JMP   012B
11BC:0126 88840001  MOV   [SI+0100],AL ┐ 80Hより小さいデータをDS：[SI+100H]
11BC:012A 46        INC   SI           ┘ に格納
11BC:012B EBE5      JMP   0112
11BC:012D 90        NOP
```

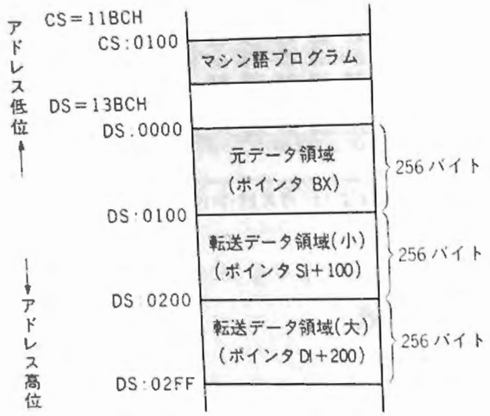


図4-3-1　メモリの使い方

このプログラムでは，処理されるデータのあるセグメントをプログラムのあるコード・セグメント（CS）より2000Hバイト後方におくことにして，データ・セグメントDSにCS+200Hの値を転送します．そのためには，まずCSの値をAXに転送し，次にAX上で200Hを加えてからDSに転送しています．このようにセグメント・レジスタに対して値をセットするには，一度汎用レジスタを介して行う必要があります．

比較される元データは，DSのオフセット0からFFH番地に置くことにし，次の256バイトが比較データより小さいデータの領域，その後の256バイトに比較データ以上のデータを格納することにします（図4-3-1）．そのために各領域中のデータを指すポインタとして，BX，SI，DIをそれぞれ使用し，SI，DIにはオフセットとして100H，200Hを加えてインダイレクト（間接）モードで参照します．

比較のための基準データは80Hとし，これをAHに代入したうえ，各領域のポインタとなるBX，SI，DIを0にします．

続いてデータの比較分類のループに入ります．その初めは，比較データ領域のポインタBXが最終データを過ぎたかの判定で，これにCMP（比較）命令が使われます．

CMP命令の表記は次のようになります．

```
CMP   DEST , SORC
```

CMP命令はDEST部とSORC部で指定されたデータを比較しますが，第3章でも述べたように，内部ではDEST部からSORC部を減算し，その結果をフラグで表しているのです．

比較できるのは，メモリーレジスタ，レジスターレジスタ，レジスターメモリ，レジスターイミディエイトデータ，メモリーイミディエイトデータの組み合せです．

ところで，CMP命令の後には必ずといってよいほど条件分岐命令が置かれますが，ここでは，JZ（Jump-if-Zero）を使い，CMP命令でBXと100Hとを比較して，結果が同じ（ゼロフラグ

が立つ)ならばプログラムの終わり(ここではNOPという何も行わない命令)へジャンプします.

比較データはBXを指標としたインダイレクトモードでALに転送したのち,これをAHの内容80Hと比較します.続いてBXの値をINC命令でインクリメントして,次のデータを指すようにします.

INC命令は,前出のDEC命令と逆に,レジスタの内容を1増加させる命令で,レジスタやメモリの内容のインクリメントにもっともよく使用される命令です.

ALとAHの比較ののち,符号なしデータのJB (Jump-Below) 命令で,ALがAHより小さければ126H番地にとび,SI+100Hの指すメモリにALの値を格納し,ALがAH以上であれば,その値をMOV命令でDI+200H番地に格納します.

格納後は,ポインタであるSIまたはDIをインクリメントしてから112H番地へ戻ります.この

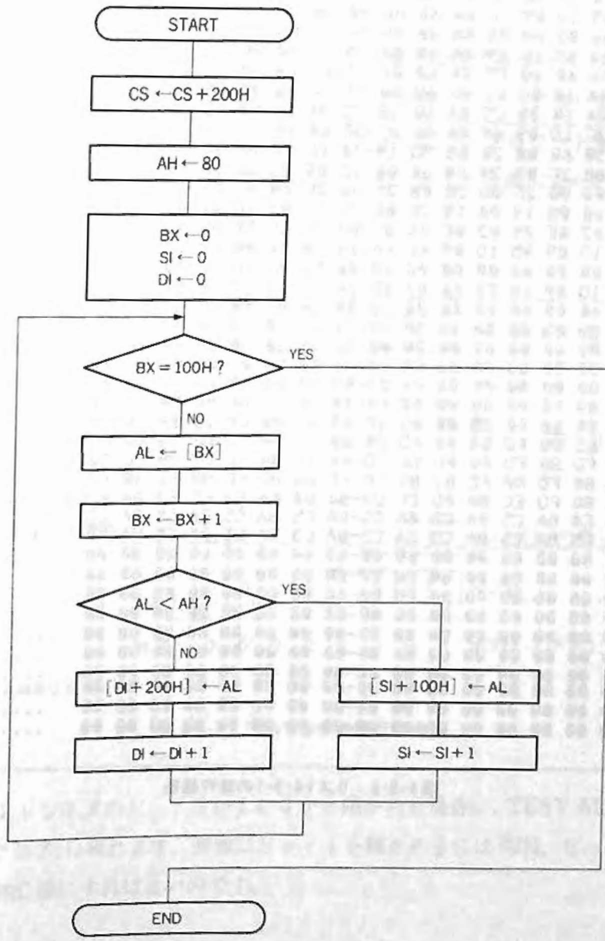


図4-3-2　フローチャート(リスト4-3-1)

とき使用するJMP命令は，オペランド部に置かれたアドレスに無条件にジャンプを行う命令です．

図4-3-2，3に，プログラムのフローチャートと実行結果を示しました．

```
-g=100,12d
AX=8000  BX=0100  CX=002E  DX=0000  SP=FFEE  BP=0000  SI=00A1  DI=005F
DS=13BC  ES=11BC  SS=11BC  CS=11BC  IP=012D   NV UP DI PL ZR NA PE NC
11BC:012D 90            NOP
-d13bc:0,2ff
13BC:0000  4C 2C 1D 09 49 0A 84 0E-E5 07 80 FD 3E 0A B4 0E   L,..I.4.e..)>.4.
13BC:0010  2B 08 80 FD D0 59 60 00-06 59 60 00 2B 08 80 FD   +..)PY`..Y`.+..)
13BC:0020  5D 19 80 FD 74 0D 80 FD-2C 08 80 FD 2C 08 80 FD   ]..)t..),..),..)
13BC:0030  52 4B 60 00 2C 08 80 FD-29 48 60 00 2C 08 80 FD   RK`.,..))H`.,..)
13BC:0040  2C 08 80 FD 2C 08 80 FD-2C 08 80 FD 58 03 00 D7   ,..),..),..)X..W
13BC:0050  2C 08 80 FD 2C 08 80 FD-2C 08 80 FD 2C 08 80 FD   ,..),..),..),..)
13BC:0060  B2 09 80 FD EF 49 60 00-42 46 60 00 84 19 80 FD   2..)oI`.BF`.....)
13BC:0070  04 18 80 FD 2B 08 80 FD-EC 27 80 FD EC 27 80 FD   ...)+..)l'.)l'.)
13BC:0080  CF 05 1D 09 D3 05 1D 09-4A 02 B4 0E 78 02 B4 0E   O...S...J.4.x.4.
13BC:0090  BA 04 E3 0D EF 07 1D 09-29 08 1D 09 07 35 1D 09   :.c.o...)....5..
13BC:00A0  E3 05 1D 09 66 8A 60 00-00 00 00 00 00 00 00 00   c...f.`.........
13BC:00B0  00 00 00 00 00 00 00 00-5A 02 E3 0D 00 00 00 00   ........Z.c.....
13BC:00C0  EA E4 05 1D 09 8A 60 00-C5 8A 60 00 C5 8A 60 00   jd....`.E.`.E.`.
13BC:00D0  C5 8A 60 00 C5 8A 60 00-C5 8A 60 00 C5 8A 60 00   E.`.E.`.E.`.E.`.
13BC:00E0  C5 8A 60 00 C5 8A 60 00-C5 8A 60 00 C5 8A 60 00   E.`.E.`.E.`.E.`.
13BC:00F0  C5 8A 60 00 C5 8A 60 00-C5 8A 60 00 C5 8A 60 00   E.`.E.`.E.`.E.`.
13BC:0100  4C 2C 1D 09 49 0A 0E 07-3E 0A 0E 2B 08 59 60 00   L,..I...>..+.Y`.
13BC:0110  06 59 60 00 2B 08 5D 19-74 0D 2C 08 2C 08 52 4B   .Y`.+.].t.,.,.RK
13BC:0120  60 00 2C 08 29 48 60 00-2C 08 2C 08 2C 08 2C 08   `.,.)H`.,.,.,.,.
13BC:0130  58 03 00 2C 08 2C 08 2C-08 2C 08 09 49 60 00 42   X..,.,.,.,..I`.B
13BC:0140  46 60 00 19 04 18 2B 08-27 27 05 1D 09 05 1D 09   F`....+.''......
13BC:0150  4A 02 0E 78 02 0E 04 0D-07 1D 09 29 08 1D 09 07   J..x.......)....
13BC:0160  35 1D 09 05 1D 09 66 60-00 00 00 00 00 00 00 00   5.....f`........
13BC:0170  00 00 00 00 00 00 00 00-00 5A 02 0D 00 00 00 00   .........Z......
13BC:0180  05 1D 09 60 00 60 00 60-00 60 00 60 00 60 00 60   ...`.`.`.`.`.`.`
13BC:0190  00 60 00 60 00 60 00 60-00 60 00 60 00 60 00 60   .`.`.`.`.`.`.`.`
13BC:01A0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
13BC:01B0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
13BC:01C0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
13BC:01D0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
13BC:01E0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
13BC:01F0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
13BC:0200  B4 E5 80 FD B4 80 FD D0-80 FD 80 FD 80 FD 80 FD   4e.)4.)P.).).).)
13BC:0210  80 FD 80 FD 80 FD 80 FD-80 FD 80 FD D7 80 FD 80   .).).).).).)W.).
13BC:0220  FD 80 FD 80 FD B2 80 FD-EF 84 80 FD 80 FD 80 FD   ).).)2.)o..).).)
13BC:0230  EC 80 FD EC 80 FD CF D3-B4 B4 BA E3 EF E3 8A E3   l.)l.)OS44:coc.c
13BC:0240  EA E4 8A C5 8A C5 8A C5-8A C5 8A C5 8A C5 8A C5   jd.E.E.E.E.E.E.E
13BC:0250  8A C5 8A C5 8A C5 8A C5-8A C5 8A C5 8A C5 8A 00   .E.E.E.E.E.E.E..
13BC:0260  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
13BC:0270  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
13BC:0280  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
13BC:0290  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
13BC:02A0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
13BC:02B0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
13BC:02C0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
13BC:02D0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
13BC:02E0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
13BC:02F0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
-
```

DSの値が変わっている

DSを指定

もとのデータ

80Hより小さいデータ

80H以上のデータ

図4-3-3　リスト4-3-1の実行結果

4-3-2　その他の比較命令

データの比較命令にはその他にCMPS命令とTEST命令があります．

CMPS命令は，ストリング命令の1つで，エクストラ・セグメント内でDIの指すメモリの内容からデータ・セグメント内のSIの指すメモリの内容を減算し，その結果をフラグ・レジスタで表します．つまりメモリーメモリ間のCMP命令であり，前述のMOVS命令と同じディレクションフラグDFの状態により，バイトデータ比較なら1ずつ，ワードデータ比較なら2ずつ，SIとDIの値が増減されます．

MOVS命令と同じく，バイトデータの比較はCMPSB，ワードデータではCMPSWと書きます．

これを使えば，2つの領域のデータがすべて同じであるかどうかが簡単に判定できるので，文字列データの比較などによく用いられます．

この命令の例として，メモリ上の2つの文字列を比較するプログラムを示します（リスト4-3-2，図4-3-4）．

比較命令のもうひとつはTEST命令です．これは一種の論理演算命令で，指定された2つのオペランドの1ビットずつの論理積（AND）をとり，その結果をフラグで表します．オペランド自身は変化しません．TEST命令は比較といってもこれまでのバイトやワード単位の数値としてのデータ比較とは異なり，データのうちの任意のビットが0か1かを調べるために使います．

図4-3-5にTEST命令の実行の様子を示します．

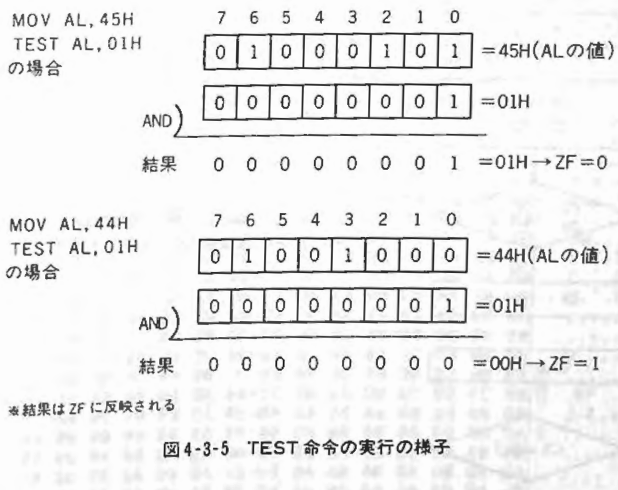


図4-3-5　TEST命令の実行の様子

このように，ALレジスタのビット0が1か0かを確かめる場合に，TEST AL, 01Hを行えば，結果はZF（ゼロフラグ）に現れます．同様にビット1を確かめるには02H，ビット3なら04Hをオペランドの SORC部にすればよいのです．

リスト 4-3-2 比較プログラム

```
11BC:0100 FC          CLD
11BC:0101 BB0000      MOV    BX,0000
11BC:0104 BE0002      MOV    SI,0200     ] DS : 200H番地からのデータとES : 300H
11BC:0107 BF0003      MOV    DI,0300     ] 番地からのデータを比較
11BC:010A B90A00      MOV    CX,000A ―― 比較するデータ数AH(10)個(バイト)
11BC:010D A6          CMPSB ――――――――― DS : [SI]の値とES : [DI]の値を比較
11BC:010E 7401        JZ     0111 ―――― 等しければ112番地へ
11BC:0110 43          INC    BX ―――――― 等しくないデータの数
11BC:0111 49          DEC    CX
11BC:0112 79F9        JNS    010D
11BC:0114 90          NOP
```

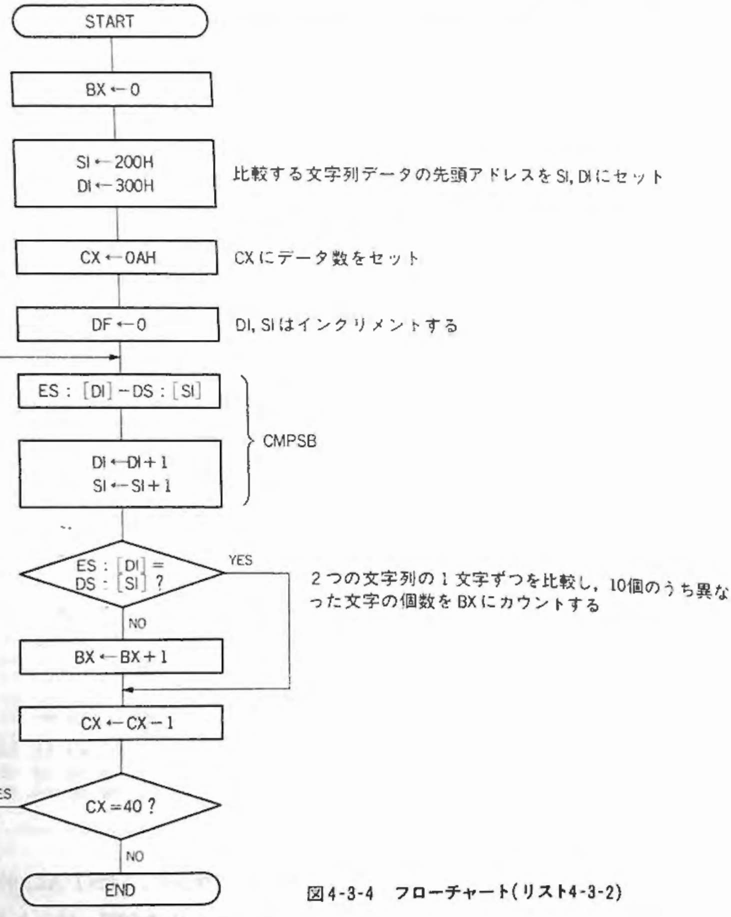


図4-3-4　フローチャート(リスト4-3-2)

TEST命令の例として，リスト4-3-1のプログラムを偶奇によるデータ分類プログラムに変えてみます．これは2進数の最下位ビットが0なら偶数，1なら奇数であることを利用したものです(リスト4-3-3，図4-3-6)．

リスト 4-3-3 偶奇判定プログラム

```
11BC:0100 8CC8        MOV   AX,CS            ┐
11BC:0102 050002      ADD   AX,0200          │CS+200HをDSに転送
11BC:0105 8ED8        MOV   DS,AX            ┘
11BC:0107 BB0000      MOV   BX,0000
11BC:010A BE0000      MOV   SI,0000
11BC:010D BF0000      MOV   DI,0000
11BC:0110 81FB0001    CMP   BX,0100
11BC:0114 7415        JZ    012B
11BC:0116 8A07        MOV   AL,[BX]
11BC:0118 43          INC   BX
11BC:0119 A801        TEST  AL,01 ───────────偶数の判定
11BC:011B 7407        JZ    0124 ────────────偶数ならば124H番地に分岐
11BC:011D 88850002    MOV   [DI+0200],AL ──DS:[DI+200H]番地にデータを格納
11BC:0121 47          INC   DI ──────────────DSを1増やす
11BC:0122 EB05        JMP   0129
11BC:0124 88840001    MOV   [SI+0100],AL ──DS:[SI+100H]番地にデータを格納
11BC:0128 46          INC   SI ──────────────SIを1増やす
11BC:0129 EBE5        JMP   0110
11BC:012B 90          NOP
```

```
-g=100,12b
        ──DSの値が変わっている
AX=1300  BX=0100  CX=002C  DX=0000  SP=FFEE  BP=0000  SI=00A4  DI=005C
DS=13BC  ES=11BC  SS=11BC  CS=11BC  IP=012B   NV UP DI PL ZR NA PE NC
11BC:012B 90            NOP
-dds:0,2ff
13BC:0000  4C 2C 1D 09 2B 08 80 FD-E5 07 80 FD 2B 08 80 FD   L,..+..)e..)+..)
13BC:0010  2B 08 80 FD D0 59 60 00-06 59 60 00 2B 08 80 FD   +..)PY`..Y`.+..)
13BC:0020  5D 19 80 FD 74 0D 80 FD-2C 08 80 FD 2C 08 80 FD   ]..)t..),..),..)
13BC:0030  52 4B 60 00 2C 08 80 FD-29 48 60 00 2C 08 80 FD   RK`.,..))H`.,..)
13BC:0040  2C 08 80 FD 2C 08 80 FD-2C 08 80 FD 58 03 00 D7   ,..),..),..)X..W
13BC:0050  2C 08 80 FD 2C 08 80 FD-2C 08 80 FD 2C 08 80 FD   ,..),..),..),..)
13BC:0060  B2 09 80 FD EF 49 60 00-42 46 60 00 84 19 80 FD   2..)oI`.BF`....)
13BC:0070  04 18 80 FD 2B 08 80 FD-EC 27 80 FD EC 27 80 FD   ...)+..)l'.)l'.)
13BC:0080  CF 05 1D 09 D3 05 1D 09-4A 02 B4 0E 78 02 84 0E   O...S...J.4.x.4.
13BC:0090  BA 04 E3 0D EF 07 1D 09-29 08 1D 09 07 35 1D 09   :.c.o...)....5..
13BC:00A0  E3 05 1D 09 66 8A 60 00-00 00 00 00 00 00 00 00   c...f.`.........
13BC:00B0  00 00 00 00 00 00 00 00-5A 02 E3 0D 00 00 00 00   ........Z.c.....
13BC:00C0  EA E4 05 1D 09 8A 60 00-C5 8A 60 00 C5 8A 60 00   jd....`.E.`.E.`.
13BC:00D0  C5 8A 60 00 C5 8A 60 00-C5 8A 60 00 C5 8A 60 00   E.`.E.`.E.`.E.`.
13BC:00E0  C5 8A 60 00 C5 8A 60 00-C5 8A 60 00 C5 8A 60 00   E.`.E.`.E.`.E.`.
13BC:00F0  C5 8A 60 00 C5 8A 60 00-C5 8A 60 00 C5 8A 60 00   E.`.E.`.E.`.E.`.
13BC:0100  4C 2C 08 80 80 08 80 08-80 D0 60 00 06 60 00 08   L,.......P`..`..
13BC:0110  80 80 74 80 2C 08 80 2C-08 80 52 60 00 2C 08 80   ..t.,..,..R`.,..
13BC:0120  4B 60 00 2C 08 80 2C 08-80 2C 08 80 2C 08 80 58   H`.,..,..,..,..X
13BC:0130  00 2C 08 80 2C 08 80 2C-08 80 2C 08 80 B2 80 60   .,..,..,..,..2.`
13BC:0140  00 42 46 60 00 84 80 04-18 80 08 80 EC 80 EC 80   .BF`........l.l.
13BC:0150  4A 02 B4 0E 78 02 B4 0E-BA 04 08 66 8A 60 00 00   J.4.x.4.:..f.`..
13BC:0160  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 5A   ...............Z
13BC:0170  02 00 00 00 00 EA E4 8A-60 00 8A 60 00 8A 60 00   .....jd.`..`..`.
13BC:0180  8A 60 00 8A 60 00 8A 60-00 8A 60 00 8A 60 00 8A   .`..`..`..`..`..
13BC:0190  60 00 8A 60 00 8A 60 00-8A 60 00 8A 60 00 8A 60   `..`..`..`..`..`
13BC:01A0  00 8A 60 00 00 00 00 00-00 00 00 00 00 00 00 00   ..`.............
13BC:01B0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
13BC:01C0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
13BC:01D0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
13BC:01E0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
13BC:01F0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
```

もとのデータ（13BC:0000〜00FF）

偶数データ（13BC:0100〜01A3）

```
13BC:0200  1D 09 2B FD E5 07 FD 2B-FD 2B FD 59 59 2B FD 5D   ..+)e.)+)+)YY+)]
13BC:0210  19 FD 0D FD FD FD 4B FD-29 FD FD FD FD 03 D7 FD   .).)))K))))))).W)
13BC:0220  FD FD FD 09 FD EF 49 19-FD FD 2B FD 27 FD 27 FD   ))).)oI.))+)')')
13BC:0230  CF 05 1D 09 D3 05 1D 09-E3 0D EF 07 1D 09 29 1D   O...S...c.o...).
13BC:0240  09 07 35 1D 09 E3 05 1D-09 E3 0D 05 1D 09 C5 C5   ..5..c...c....EE
13BC:0250  C5 C5 C5 C5 C5 C5 C5 C5-C5 C5 C5 C5 00 00 00 00   EEEEEEEEEEEE....
13BC:0260  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
13BC:0270  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
13BC:0280  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
13BC:0290  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
13BC:02A0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
13BC:02B0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
13BC:02C0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
13BC:02D0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
13BC:02E0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
13BC:02F0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
-
```

奇数データ

図4-3-6 リスト4-3-3の実行結果

このTEST命令は，また，ハードウェアをコントロールする際の，周辺デバイスのステータス(状態)フラグの内容をテストするときなどにもよく使われます．

4-3-3 フラグと条件分岐

これまでのサンプル・プログラムでは，いくつかの条件分岐命令（JZ等）を使ってループやプログラムの流れを変更してきましたが，ここでこれらについてまとめておきましょう．

条件分岐の条件とは，フラグ・レジスタの状態を示しています．このフラグ・レジスタの内容は，比較命令や後述の演算命令，INC，DEC命令等により，様々に変化します．

条件分岐命令は，そのフラグ・レジスタの中から，自分に関係のあるフラグを見て，分岐するかどうかを決めるのです．各命令によるフラグの変化は巻末のインストラクション表を参照してください．

ところで，データの大小比較のところで，マシン語には同じ作業を行うのにも，データを符号つきとみなすか，符号なしとみなすかによって，異なる命令を使用しなければならないものがあると述べましたが，ここで解説している条件分岐命令もそのひとつです．ここで，条件分岐の命令を整理しておきましょう（図4-3-7）．

符号つきデータといっても，メモリ内の数値に，正負の符号がついているわけではないので，プログラム中で，処理結果を符号つきの値として条件分岐したい場合は，その正負を示すサインフラグSFと，オーバーフローフラグOVとの関係を分岐の条件としている命令（JNS等）を使用するのです．

条件分岐命令の中でひとつだけフラグ・レジスタを参照しない命令があります．これはCXレジスタの値が0のときにのみ分岐する命令で，JCXZ命令といいます．この命令を使って，メモリを0～FHで埋めるプログラム例をリスト4-3-4に示します（図4-3-8）．

①条件分岐命令一覧

命　令	分岐が起きる条件	命　令	分岐が起きる条件
JA/JNBE	(CF=0)∧(ZF=0)	JNZ/JNE	ZF=0
JBE/JNA	(CF=1)∨(ZF=1)	JS	SF=1
JNB/JAE/JNC	CF=0	JNS	SF=0
JB/JNAE/JC	CF=1	JPE/JP	PF=1
JG/JNLE	(SF⊻OF=0)∧(ZF=0)	JPO/JNP	PF=0
JLE/JNG	(SF⊻OF=1)∨(ZF=1)	JO	OF=1
JGE/JNL	SF⊻OF=0	JNO	OF=0
JL/JNGE	SF⊻OF=1	JCXZ	CX=0
JZ/JE	ZF=1		

※∧, ∨, ⊻は論理記号で,
　∧(and) ……………「かつ」を表す
　∨(or) ………………「または」を表す
　⊻(exclusive or)……一方のみ真のとき真, 両方真または偽のとき偽となる
　　　　　　　　　　(フラグの内容が1のとき真, 0のとき偽とする)

②データの大小による分岐命令

分類	符号なし数値用	符号つき数値用
D=S	JE(JZ)	JZ(JE)
D≠S	JNE(JNZ)	JNZ(JNE)
D>S	JA(JNBE)	JG(JNLE)
D≦S	JNA(JBE)	JNG(JLE)
D≧S	JAE(JNB, JNC)	JGE(JNL)
D<S	JB(JNAE, JC)	JL(JNGE)

※D, Sはオペランドでそれぞれディスティネーション, ソースを指す

③フラグによる分岐命令

命　令	分岐が起きる条件
JZ(JE)	ZF=1
JNZ(JNE)	ZF=0
JS	SF=1
JNS	SF=0
JP(JPE)	PF=1
JNP(JPO)	PF=0
JC(JB, JNAE)	CF=1
JNC(JNB, JAE)	CF=0
JO	OF=1
JNO	OF=0

④レジスタの値による分岐命令

命　令	分岐が起きる条件
JCXZ	CX=0

図4-3-7　条件分岐命令の種類とフラグ

リスト 4-3-4　JCXZ命令の使用例

```
11BC:0100 BE0000      MOV    SI,0000
11BC:0103 81FE0001    CMP    SI,0100
11BC:0107 7411        JZ     011A
11BC:0109 B91000      MOV    CX,0010
11BC:010C B000        MOV    AL,00
11BC:010E 88840002    MOV    [SI+0200],AL ――― DS:[SI+200H]にALの値を転送
11BC:0112 46          INC    SI ―――――――― SIをインクリメント
11BC:0113 FEC0        INC    AL ―――――――― ALをインクリメント
11BC:0115 49          DEC    CX ―――――――― CXをデクリメント
11BC:0116 E3EB        JCXZ   0103 ――――――― CXが0ならば103Hに分岐
11BC:0118 EBF4        JMP    010E
11BC:011A 90          NOP
```

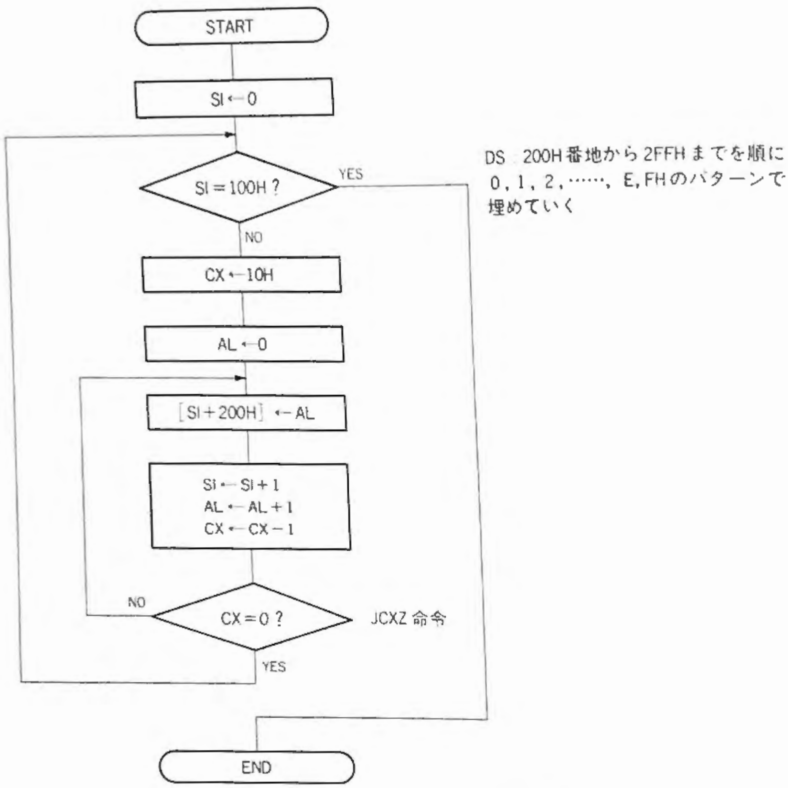


図4-3-8　フローチャート(リスト4-3-4)

ところで，これまで条件分岐命令の使用に際しては，何の断りもせずに使用してきましたが，実はこれらの分岐命令によって指定可能な分岐先アドレスは，その分岐命令の次のアドレスから数えて−128～127バイトまでの範囲に限られています（図4-3-9）.

マシン語オブジェクト（16進数）には，分岐命令に続いてこの距離が16進1バイトで書かれているのです.

ところで，デバッガのAコマンドでプログラムを入力するときには，ジャンプ先のアドレスを直接指定していました．これはデバッガが，入力された値から，この相対的な距離計算を行っているためです．ですから，試しにAコマンドを使ってアドレス1000H番地あたりから，

```
−A1000
××××：1000　JZ　100
```

と入力するとエラーが起きます．これは，アドレス1002Hからの分岐の距離が限界を超えたために起きたのです.

このように分岐先をその命令のあるアドレスからの距離で指定する方法を相対分岐といいます．相対分岐の計算方法を図4-3-10に示します.

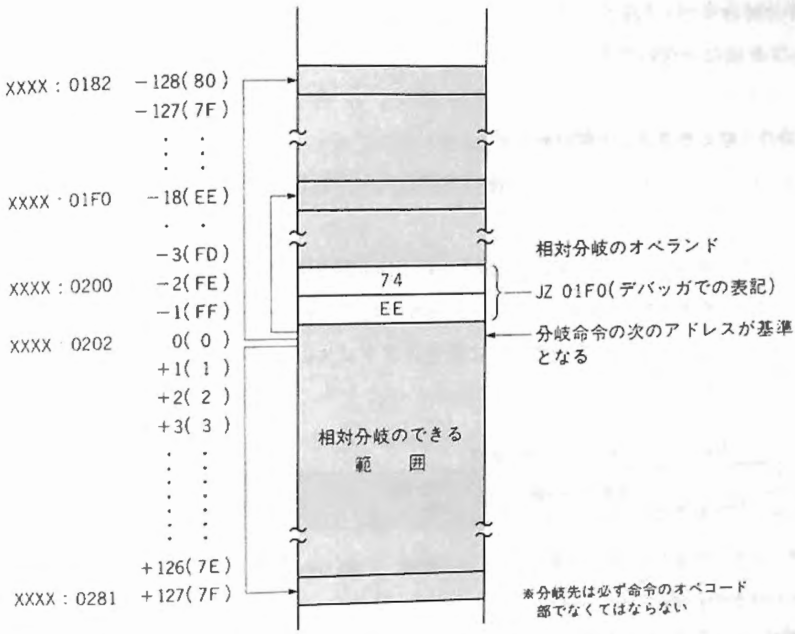


図4-3-9　相対分岐の範囲

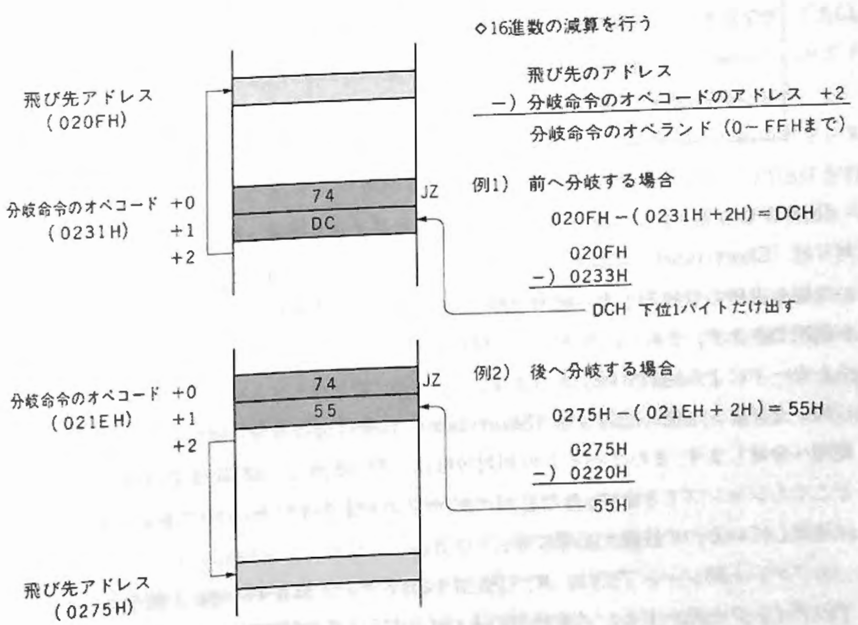


図4-3-10　相対分岐の計算例

8086の条件分岐命令が－128から127までしかジャンプできないのは，距離を示すオペランドが1バイトしかとれないからです．

4-3-4 無条件分岐とセグメント間ジャンプ

さて，分岐命令で－128～127バイトより遠いアドレスへ分岐したい場合はどうすればよいのでしょうか．

8086の分岐命令には，セグメント内，およびセグメント間をも自由に分岐できる無条件分岐命令JMPがあります．無条件分岐命令とは，文字どおり分岐のための条件を持たない命令のことで，この命令が実行されると，フラグ等に無関係に指定のアドレスに実行を分岐します．JMP命令のアドレッシングを，**表4-3-1**に示します．

表4-3-1　JMP命令のアドレッシングと表現

モード	セグメント内	相対・絶対の別	セグメント間	相対・絶対の別
イミディエイト	JMP オフセット値 (Short or Near)	相対	JMP セグメント値：オフセット値	相対
レジスタ	JMP レジスタ	絶対		
ダイレクト	JMP［オフセット値］	絶対	JMP FAR［オフセット値］	絶対
インダイレクト	JMP［Reg*］	絶対	JMP FAR［Reg*］	絶対

＊インダイレクトモード用レジスタ(4-2参照)

JMP命令のアドレッシングモードは，同一セグメント内は，条件分岐命令と同じ1バイトの相対分岐（Short-label）に加え，2バイトの相対分岐（Near-label）が行え，またオペランドに置いた値を直接の分岐先とする絶対分岐モードとして，ダイレクトモード，インダイレクトモードが使用できます．さらに，セグメント間の分岐用に，ダイレクトモード(Far-label)，インダイレクトモードによる絶対分岐があります．

同一セグメント内の相対分岐（Short-label）は条件分岐命令と同じで，－128から＋127までの範囲へ分岐します．また2バイトの相対分岐は－32768から＋32767までの同一セグメント内ならどこでもジャンプできます．ただし，1つのセグメントのオフセット・アドレスFFFFHと，0000Hは連続しているので注意が必要です．

セグメント間ジャンプでは，オペランドにセグメント値とオフセット値を4バイトで直接指定するダイレクトモードと，メモリ上の4バイトにジャンプ先アドレスをオフセット値，セグメント値の順で格納し，これの先頭アドレスをレジスタで指定することで，間接的に分岐先を指定するインダイレクトモードがあります．

4-3-5 サブルーチンコールとスタック

無条件分岐命令JMPの他に8086の全アドレスのジャンプの行える命令にCALL (Call-procedure) 命令があります．しかし，この命令はただのJMPではなく，サブルーチンコールのための命令です．サブルーチンとはご存知の通り何度か使用される処理をひとつにして，メインルーチンからその都度呼び出すもので，処理終了後にはメインルーチンへ戻ることによってプログラムを短くし，見通しをよくします．

BASICではサブルーチンはGOSUB文で飛び先を指定してコールし，RETURN文で戻りますが，このとき戻るための行番号は指定しません．これは，BASICが，内部で呼出し元の番地を憶えているからです．

マシン語プログラムのサブルーチンコールも同じですが，呼出しにはCALL命令を使い，戻るにはサブルーチン側でRET (Return-from-procedure) 命令を行います．このときサブルーチンを呼び出したCALL命令の次の命令のアドレス2バイトが，スタック上にPUSH (退避) されます（セグメント内CALLの場合）．このアドレスは，実はインストラクション・ポインタIPの値です．スタックは第2章で説明した通り，LIFOのメモリで，そのアクセスされるアドレスはスタック・ポインタSPとスタック・セグメント・レジスタSSにより，常に示されています．

これに対し，RET命令はスタックからCALLでしまわれた戻り先のアドレスをPOP (引き出す) して，インストラクション・ポインタIPに転送します．これにより，サブルーチンから戻ると，呼出し側のメインルーチンのCALLの次の命令からプログラムが実行されるのです．CALLは，JMPと同じく2バイト・オフセット（セグメント内），および4バイト・オフセット（セグメント間）のダイレクトモードと，インダイレクトモードのアドレッシングを持っています．

ところで，RET命令は，命令実行時のSPの指すアドレスのデータを，復帰のためのアドレスとして使用するため，もしCALL命令でPUSHした後にPOPが行われたり，別のPUSHが行なわれて，そのままだったり，またSPの値が書き変えられたりした場合には，正しいアドレスへ戻れなくなってしまいますので注意が必要です．

CALL，RET以外にスタックを積極的に使用するための命令として，PUSH (Push-word-to-stack) 命令とPOP (Pop-word-off-stack) 命令があります．これらは，オペランド部に置かれたワードデータ (レジスタ，2バイトメモリ等) をスタックに出し入れする命令で，データの転送とともに，PUSHでスタック・ポインタSPの値が+2だけ，POPで−2だけ増減し，最後にPUSHされたデータのあるアドレスを示します．

その一般的な使い方は，サブルーチンの際のレジスタの内容の退避です．

これは，メインルーチンが汎用レジスタなどに必要なデータを入れたままサブルーチンコールをした場合，サブルーチン側がそのレジスタの内容を変更してしまうと，メインルーチンへ戻った時，メインルーチンで正常な作業ができなくなるという情況を防ぐために使用するものです．これは，サブルーチン内で使用するレジスタについてはあらかじめそのサブルーチンの先頭でPUSHし，サブルーチンでの処理が終って，RET命令で戻る前に，POP命令によって，レジスタ内容を

復帰することで行います．

ここで気をつけるのは，PUSH，POP の数を合わせることと，PUSH したレジスタの順番と POP するレジスタの順を逆，つまり先にPUSHしたものを最後にPOPするということです（図 4-3-11）．

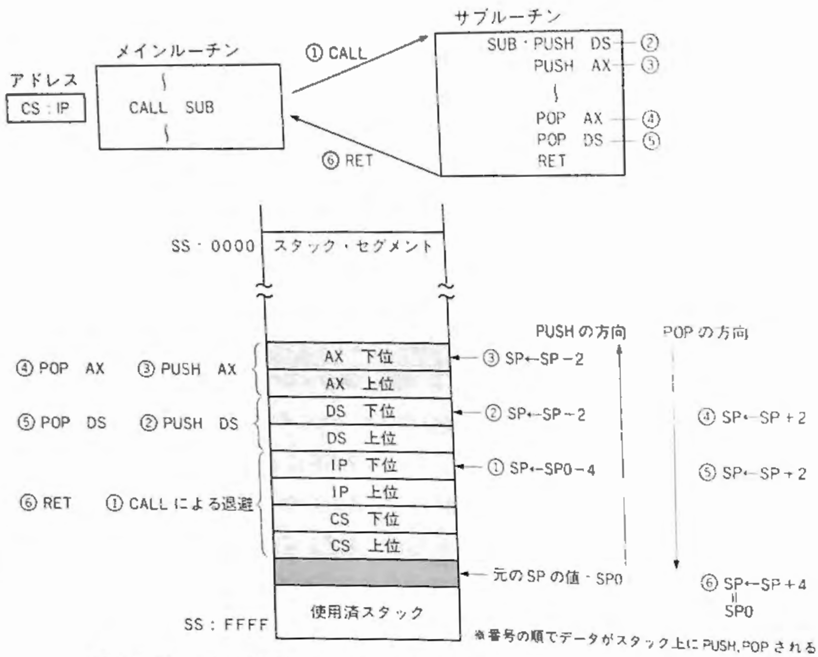


図 4-3-11　CALL 命令とスタックへの PUSH，POP（セグメント間 CALL の場合）

マシン語プログラムでサブルーチンを使った例をリスト 4-3-5 に示します．このプログラムはリスト 4-3-1 をサブルーチン化したもので，データ領域を 256 バイトとし，メインルーチンで BX にそのデータ領域のアドレス・オフセット，AH に比較データをセットしてサブルーチンコールを行います．サブルーチン側では，内部で使用する BX，AX の内容を PUSH したのち，大小分類処理を行い，POP でレジスタの内容を復帰してメインルーチンに戻ります（図 4-3-12）．

リスト 4-3-5　サブルーチンコールの例

```
11BC:0100 BB0002          ┌MOV    BX,0200 ―――― サブルーチンに渡す引数
11BC:0103 B480            │MOV    AH,80 ―――― サブルーチン中で比較する値
11BC:0105 E80900   メイン │CALL   0111 ―――― サブルーチンのコール
11BC:0108 83C310   ルーチン│ADD    BX,+10 ―――― サブルーチンに渡す引数(入力時はADD BX,10)
11BC:010B B440            │MOV    AH,40 ―――― サブルーチン中で比較する値
11BC:010D E80100          └CALL   0111 ―――― サブルーチンのコール
11BC:0110 90               NOP
11BC:0111 1E              ┌PUSH   DS      ┐
11BC:0112 50              │PUSH   AX      │サブルーチン中で使用するレジスタの退避
11BC:0113 53              │PUSH   BX      ┘
11BC:0114 8CC9            │MOV    CX,CS   ┐
11BC:0116 03CB            │ADD    CX,BX   │CS+BXを処理されるデータのあるセグメントDSとする
```

```
11BC:0118 8ED9        MOV  DS,CX
11BC:011A BB0000      MOV  BX,0000
11BC:011D BE0000      MOV  SI,0000
11BC:0120 BF0000      MOV  DI,0000
11BC:0123 81FB0001    CMP  BX,0100
11BC:0127 7415        JZ   013E
11BC:0129 8A07        MOV  AL,[BX]
11BC:012B 43          INC  BX
11BC:012C 3AC4        CMP  AL,AH          大小比較ルーチン
11BC:012E 7207        JB   0137
11BC:0130 88850002    MOV  [DI+0200],AL
11BC:0134 47          INC  DI
11BC:0135 EB05        JMP  013C
11BC:0137 88840001    MOV  [SI+0100],AL
11BC:013B 46          INC  SI
11BC:013C EBE5        JMP  0123
11BC:013E 5B          POP  BX
11BC:013F 58          POP  AX             レジスタの復帰
11BC:0140 1F          POP  DS
11BC:0141 C3          RET ─────────────── メインルーチンに戻る
```

サブルーチン

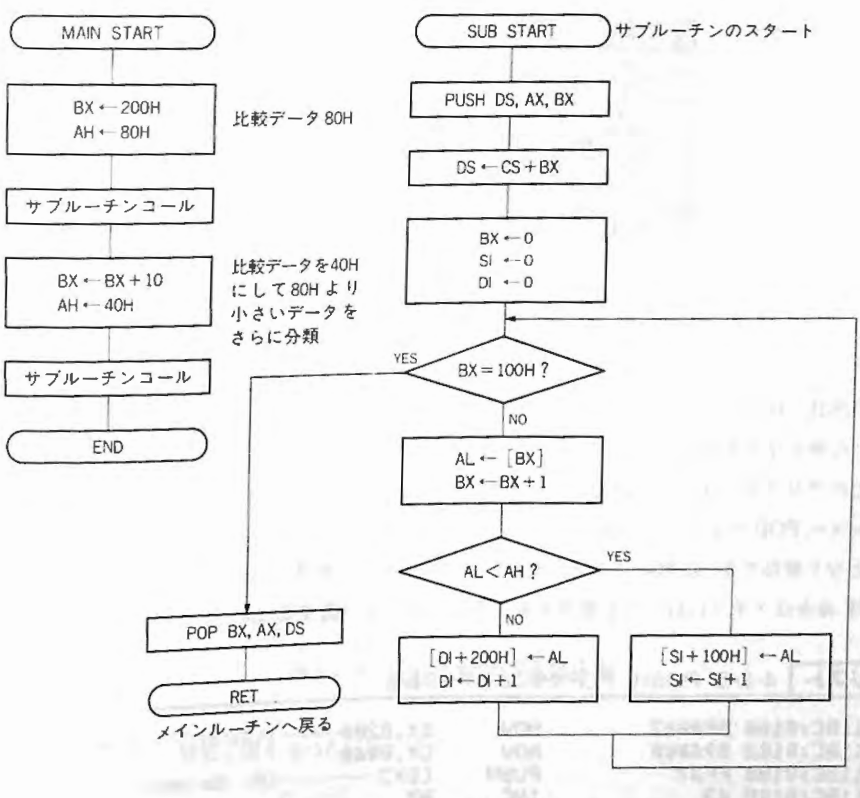


図4-3-12 フローチャート(リスト4-3-5)

ところで、PUSH，POP の用法としては，レジスタ内容の保持だけでなく，メインルーチンからサブルーチンへデータを渡すのに使用することもできます．つまり，メインルーチンで引数データを必要なだけ PUSH しておき，サブルーチンでは同じ数だけ POP を行ってこれを受けとるか、またはスタックを直接アクセスしてデータを引き出すかするわけです（図 4-3-13）．

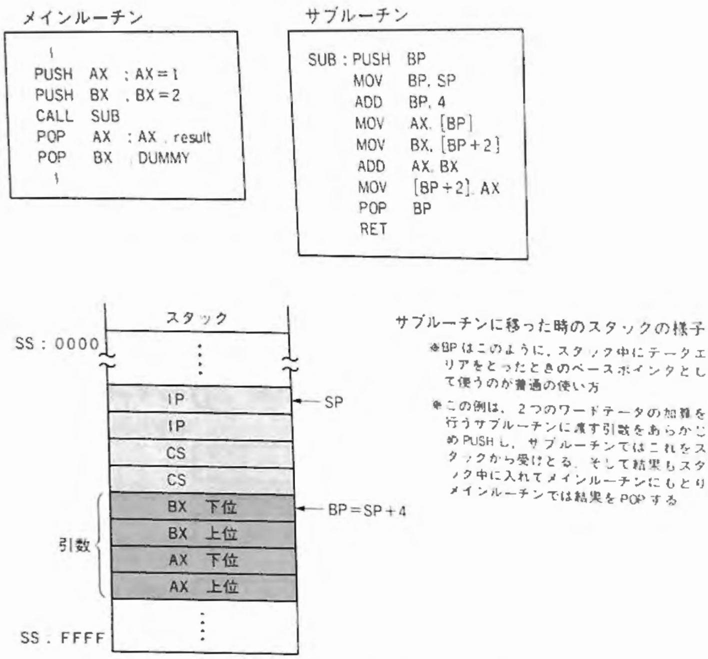


図 4-3-13　スタックを使った引数の受け渡し

PUSH，POP 命令を使えば，MOVS 命令を使わずにデータのブロック転送が行えます．この例をリスト 4-3-6 に示しました．

このプログラムは，ある領域のデータをすべてスタック上に PUSH し，それを別の転送先のアドレスへ POP するもので，MOVS 命令のように 1 命令では行えませんが，SI と DI を占有されることなく使用でき，汎用レジスタも使わないで済むなどの利点があります．このように，PUSH，POP 命令はメモリに対しても使用することができます（図 4-3-14）．

リスト 4-3-6　PUSH，POP命令によるデータ転送

```
11BC:0100 BB0002        MOV     BX,0200 ──── PUSHされるデータの開始番地
11BC:0103 B94000        MOV     CX,0040
11BC:0106 FF37          PUSH    [BX] ─────── DS：[BX]の値をスタックにPUSH
11BC:0108 43            INC     BX      ┐
11BC:0109 43            INC     BX      ┘ BXの値を2へらす
```

```
11BC:010A E2FA          LOOP    0106 ──── CXを1ずつ減じながら0になるまで106Hへ分岐
11BC:010C BBFE02        MOV     BX,02FE ── POPされたデータの格納される開始番地
11BC:010F B94000        MOV     CX,0040
11BC:0112 8F07          POP     [BX] ───── スタックからDS:[BX]にデータをPOP
11BC:0114 4B            DEC     BX      ┐
11BC:0115 4B            DEC     BX      ┘ BXの値を2増やす
11BC:0116 E2FA          LOOP    0112 ──── CXを減じながら0になるまで112Hへ分岐
11BC:0118 90            NOP
```

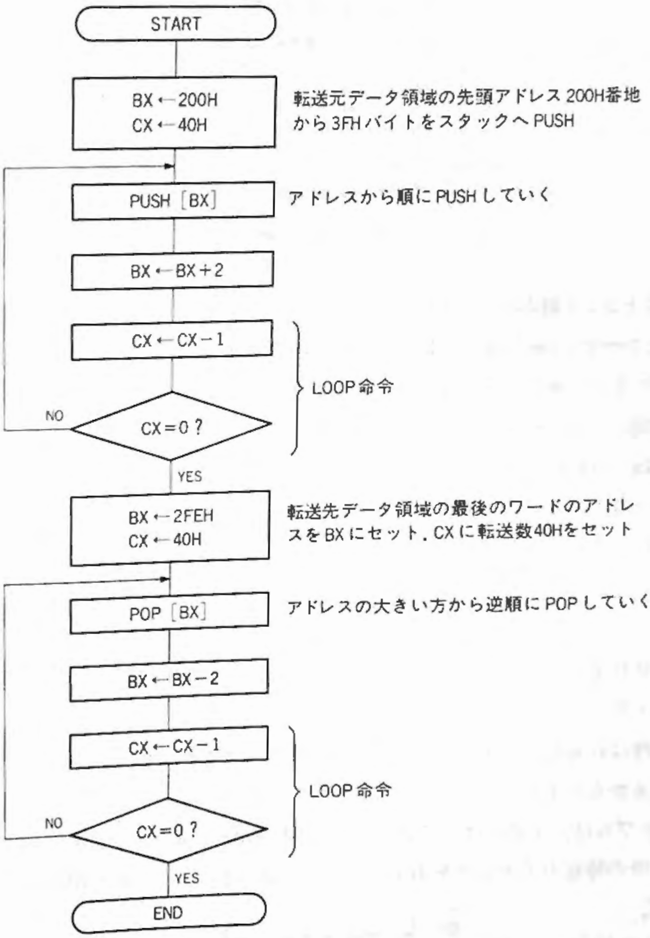


図4-3-14　フローチャート(リスト4-3-6)

プログラムの実行結果を図4-3-15に示します．

```
-g=100,118

AX=0000  BX=027E  CX=0000  DX=F500  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=11BC  ES=11BC  SS=11BC  CS=11BC  IP=0118   NV UP DI PL NZ NA PE NC
11BC:0118 90            NOP
-d200,2ff
11BC:0200  00 01 02 03 04 05 06 07-08 09 0A 0B 0C 0D 0E 0F   ................
11BC:0210  10 11 12 13 14 15 16 17-18 19 1A 1B 1C 1D 1E 1F   ................
11BC:0220  20 21 22 23 24 25 26 27-28 29 2A 2B 2C 2D 2E 2F    !"#$%&'()*+,-./
11BC:0230  30 31 32 33 34 35 36 37-38 39 3A 3B 3C 3D 3E 3F   0123456789:;<=>?
11BC:0240  40 41 42 43 44 45 46 47-48 49 4A 4B 4C 4D 4E 4F   @ABCDEFGHIJKLMNO
11BC:0250  50 51 52 53 54 55 56 57-58 59 5A 5B 5C 5D 5E 5F   PQRSTUVWXYZ[¥]^_
11BC:0260  60 61 62 63 64 65 66 67-68 69 6A 6B 6C 6D 6E 6F   `abcdefghijklmno
11BC:0270  70 71 72 73 74 75 76 77-78 79 7A 7B 7C 7D 7E 7F   pqrstuvwxyz{¦}~.
11BC:0280  00 01 02 03 04 05 06 07-08 09 0A 0B 0C 0D 0E 0F   ................
11BC:0290  10 11 12 13 14 15 16 17-18 19 1A 1B 1C 1D 1E 1F   ................
11BC:02A0  20 21 22 23 24 25 26 27-28 29 2A 2B 2C 2D 2E 2F    !"#$%&'()*+,-./
11BC:02B0  30 31 32 33 34 35 36 37-38 39 3A 3B 3C 3D 3E 3F   0123456789:;<=>?
11BC:02C0  40 41 42 43 44 45 46 47-48 49 4A 4B 4C 4D 4E 4F   @ABCDEFGHIJKLMNO
11BC:02D0  50 51 52 53 54 55 56 57-58 59 5A 5B 5C 5D 5E 5F   PQRSTUVWXYZ[¥]^_
11BC:02E0  60 61 62 63 64 65 66 67-68 69 6A 6B 6C 6D 6E 6F   `abcdefghijklmno
11BC:02F0  70 71 72 73 74 75 76 77-78 79 7A 7B 7C 7D 7E 7F   pqrstuvwxyz{¦}~.
```

（11BC:0200～0270：もとのデータ　11BC:0280～02F0：転送されたデータ）

図4-3-15　リスト4-3-6の実行結果

4-3-6　ソフトウェア割込みとベクタ

プログラム(サービス)実行中のCPUに外部から働きかけて，そのサービスの実行を中断させ，別のサービスの実行を優先させることを，ハードウェア割込み，または外部割込みといいます．ハードウェア割込みの詳細は第7章で解説しますが，プログラム中の命令によってハードウェア割込みで引き起こされる一連の作業と同様な処理を行なわせようというのが，ここで解説するソフトウェア割込み，あるいは内部割込みといわれるものです．

このソフトウェア割込みには，INTまたはINTO命令が使用され，

```
INT x
```

のように使用されます．ここでxは，メモリ空間の最下位0～3FFHにある，4バイトごとに区切られたデータ・テーブルの何番目を使用するかということを示す数値で，ベクタ番号，または，単にベクタと呼ばれるものです．INTOの場合は数値を付けずに用いますが，これはベクタが4に固定されているからです．

データ・テーブルは，正式にはベクタ・テーブルと呼ばれ，4バイトに区切られたひとつひとつには，メモリ中の特定のアドレスを示すオフセット値とセグメント値が図4-3-16のように格納されています．

INT命令を実行すると，CS，IP，およびフラグ・レジスタがPUSHされ，ベクタ番号に対応したベクタテーブル中の4バイトのアドレス(セグメントとオフセット)がCS, IPにセットされ，そのアドレスに実行が分岐することになります．

INT命令によるプログラム実行の分岐で，もとに戻るにはIRET (Interrupt-Return) 命令を用います．これはRET命令の動作に加えて，INT命令実行時にPUSHされたフラグ・レジスタ

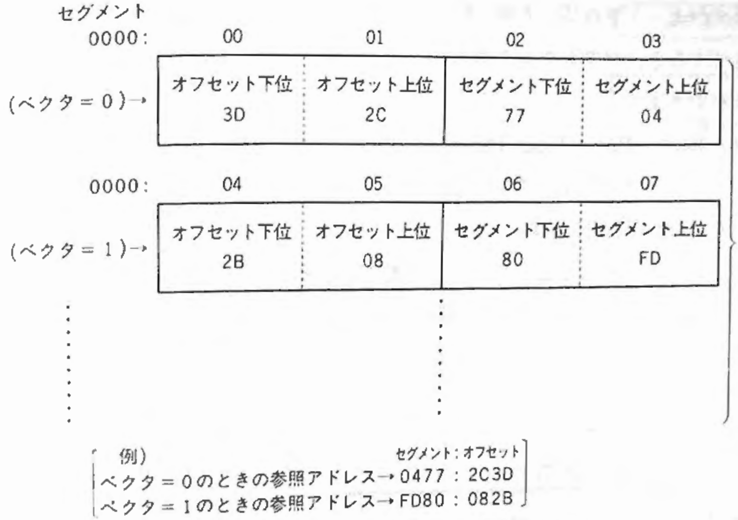


図4-3-16　割込みベクタテーブルの内容例

の値も合せて POP します．

割込みに利用されるベクタ番号は，0~FFH (255) までの値をとり得ますから，ベクタ・テーブルには，その4倍の0~3FFHバイト分があてられているのです．

PC-9801 では，ここは RAM になっているので，ユーザーが自作のルーチンに分岐するようにテーブルを書き変えることも可能です．INT 命令による割込みのようすを図 4-3-17 に示します．

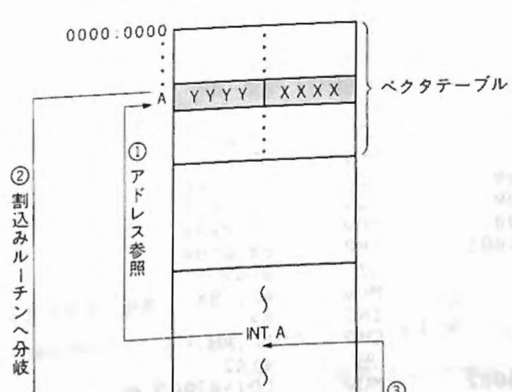


図4-3-17　INT 命令の動作

INT 命令は，ハードウェアによる割込みとは異なり，プログラム中の INT 命令のある所でしか割り込むことはできません．例外は，INTO，1で，これらはそれぞれ，0で除算を行ったとき，

トレース・フラグがセットされている時に割込みがかかります。

このように見てくると、ソフトウェア割込みといっても、事実上は一種のテーブル参照分岐命令と考えることができます。

PC-9801では、BIOS (Basic-Input-Output-System) のコールにINT命令を使うようになっており、ユーザーがベクタテーブルにアドレスをセットして使用できるのは、ベクタ番号40H～7FH、ベクタ・アドレス100H～1FFHです。

リスト4-3-7にINT命令によるソフトウェア割込みの例を示します(図4-3-18)。

このプログラムは、リスト4-3-1のデータ分類プログラムをINTの40Hでよぶプログラムです。

メインルーチンでは、ベクタテーブルに割込み処理ルーチンのアドレスをセットして、INT 40Hを実行します。

リスト 4-3-7 INT命令の使用例

```
                              (メインルーチン)
11BC:0100 1E          PUSH  DS            DSの値をスタックに退避
11BC:0101 8CC8        MOV   AX,CS
11BC:0103 BB0000      MOV   BX,0000       DSを0にする
11BC:0106 8EDB        MOV   DS,BX
11BC:0108 BE0001      MOV   SI,0100       INT40Hのベクタテーブルのアドレス
11BC:010B 894402      MOV   [SI+02],AX    セグメント値のセット
11BC:010E B81C01      MOV   AX,011C       オフセット値のセット
11BC:0111 8904        MOV   [SI],AX
11BC:0113 1F          POP   DS            DSの値をもとに戻す
11BC:0114 BB2000      MOV   BX,0020       データ領域のCSからのオフセット
11BC:0117 B47F        MOV   AH,7F         INTの実行
11BC:0119 CD40        INT   40
11BC:011B 90          NOP
                              (サブルーチン)
11BC:011C 1E          PUSH  DS            レジスタの退避
11BC:011D 50          PUSH  AX
11BC:011E 53          PUSH  BX
11BC:011F 8CC9        MOV   CX,CS         CS+BXをDSの値にする
11BC:0121 03CB        ADD   CX,BX
11BC:0123 8ED9        MOV   DS,CX
11BC:0125 BB0000      MOV   BX,0000       大小比較ルーチン
11BC:0128 BE0000      MOV   SI,0000
11BC:012B BF0000      MOV   DI,0000
11BC:012E 81FB0001    CMP   BX,0100
11BC:0132 7415        JZ    0149
11BC:0134 8A07        MOV   AL,[BX]
11BC:0136 43          INC   BX
11BC:0137 3AC4        CMP   AL,AH
11BC:0139 7207        JB    0142
11BC:013B 88850002    MOV   [DI+0200],AL
11BC:013F 47          INC   DI
11BC:0140 EB05        JMP   0147
11BC:0142 88840001    MOV   [SI+0100],AL
11BC:0146 46          INC   SI
11BC:0147 EBE5        JMP   012E
11BC:0149 5B          POP   BX            レジスタの復帰
11BC:014A 58          POP   AX
11BC:014B 1F          POP   DS
11BC:014C CF          IRET                割込みルーチンからの復帰
```

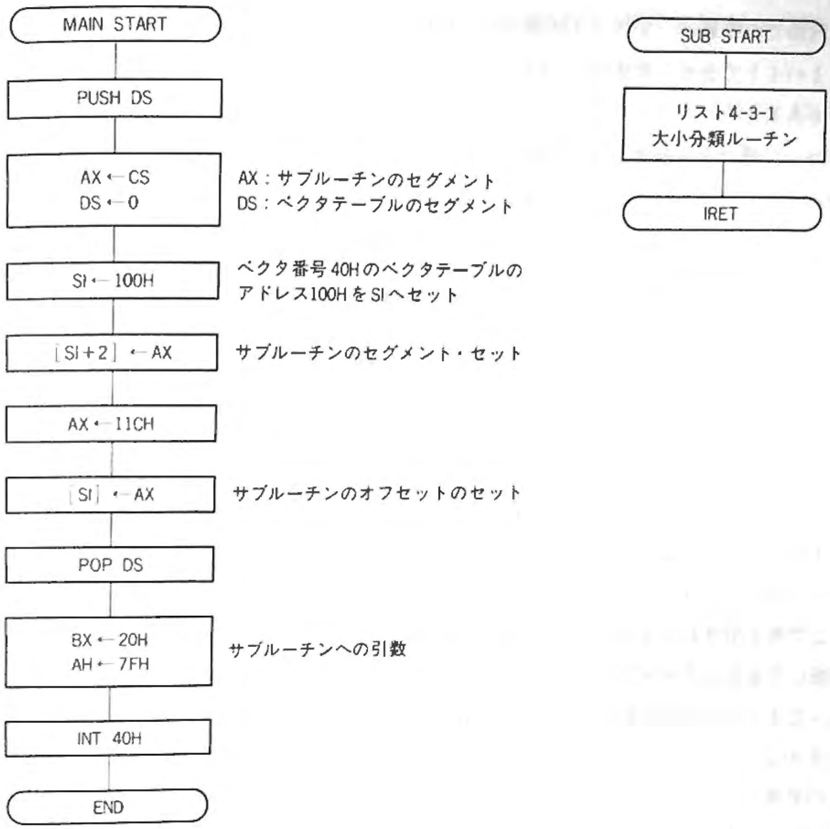


図4-3-18 フローチャート(リスト4-3-7)

4-4 データの演算とフラグ

4-4-1 符号つき演算と符号なし演算

前節までは，データの転送を中心に話を進めてきましたが，それだけでは，8086の計算機としての能力は発揮できません．ここでは，マシン語による演算について解説しましょう．

これまで，マシン語で扱う数値データについて，あまり厳格な定義をしてきませんでした．ここで少しその点に触れてみます．

マシン語で扱う数値は，例外もありますが普通はバイト単位で使用されます．8086で一度に扱えるのは，1バイトデータか2バイトデータで，それ以上の数値となると，いくつかに区切って処理する必要があります．大きな数の扱いについては後で述べるとして，このバイトデータについて基本的な点にふれておきましょう．

① 内部での演算は，すべて2進数で行われること

② 1バイトで表せる数値は0～FFH (255), 2バイトで表せる数値は0～FFFFH (65535) までであること

③ マシン語による計算では，結果がそれ以上になる場合，あふれた部分は切り捨てられ，下位の1または2バイトのみが結果となること

④ バイトデータの構造は循環していて，0の下にはFFH（2バイトデータならFFFFH）が続いているように演算されること

⑤ ただし，キャリーと呼ばれる1ビットのフラグが，データの最上位ビットのさらに上位にあって，結果がバイトデータにおさまらなかった場合の，桁上がり（キャリー）や桁下がり（ボロー）はそのビットを立てることで，検出できること

以上が，CPUによる演算の基本で，正の整数を扱うのであれば，あまり問題は起きません．しかしここで困ることは，負の数の表現方法がないことです．メモリ中のデータには，正負の符号をつけることはできないので，同じ数値を符号なしのデータと見るか，符号つきのデータと見るかをユーザーが区別しなければなりません．

そこで考え出されたものが，2の補数表現です．

簡単にするため1バイト（8ビット）データについて話を進めますが，2の補数表現とは，④に述べたようにCPU内部で0の下にFFHが続くことを利用して，これを－1と見ることで，図**4-1-1**のようにバイトデータを－128～127（80H～7FH）までの数値に対応させる表現です．

この表現のよい点は，0～127までの数値は7ビットで表現でき，さらに8ビット目（MSBと呼ぶ）が0のときは正の値，1の時は負の値となるので，これを正，負の符号と見ることができることと，結果が－128～127の間に入っている限り，上で述べた①～⑤の符号なし演算の処理に問題なく適応できるということです．

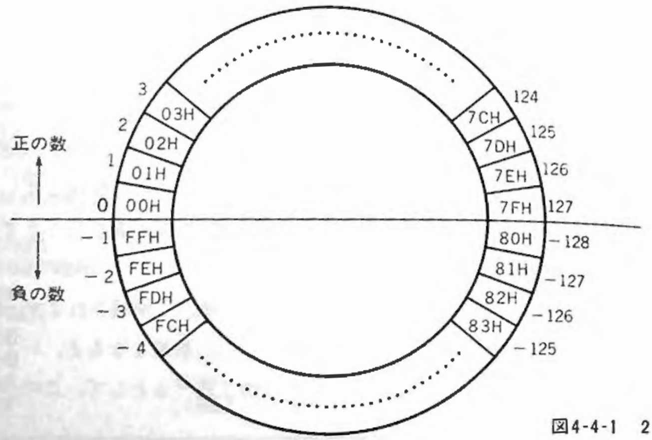


図4-4-1　2の補数表現

サインフラグSFはこのMSBの状態を示すもので，演算の結果MSBが1になる(80H以上になる）ときに1になるようになっています．

ただし，この表現にも問題点があります．ひとつは人間にとってわかりにくいことで，たとえば，−21H(−33）はどう表すのか，C9Hの値はいくつなのか，少し悩むと思います．

これには，図4-4-2のような算法が用いられます．

① −21H(−33)の2の補数表現

i）−21Hの符号をとり，2進数になおす ──→ 0010 0001 (2 1)

ii）各ビットを反転する ──→ 11011110

iii）1を加える ──→ 1101 1111 (D F)

iv）16進に戻す ──→ DFH

② 2の補数表現によるC9Hの値

i）C9Hを2進数になおす ──→ 1100 1001 (C 9)

ii）MSBが1なので負の数

iii）1を引く ──→ 11001000

iv）各ビットを反転する ──→ 0011 0111 (3 7)

v）16進に戻す ──→ 37H

vi）負の数なので−符号をつける ──→ −37H(−55)

図4-4-2　2の補数表現の算法

これ以上に問題となるのが，演算結果が−128〜127の範囲を超えた時に，正しい結果を生じないことです．

たとえば，127(7FH)に1を加えると，本当ならば128にならなければなりませんが，結果は，−128(80H)になってしまいます．もちろんこの80Hは，符号なしの演算と見なせば正しいわけで，CPUが間違ったわけではありません．ただ，ここで用いている2の補数表現による1バイトデータの演算では，結果を−128〜127の間でしか保障しない（あるいは対応できない）のです．したがって，範囲を超える結果が出ることが予想される場合は，扱うデータを2バイトや4バイトデータにしておくか，2の補数表現専用の演算命令（IMUL等）を使用して，範囲が超えた時にプログラム中で自動的にデータ長を変更するようにしておく必要があります．

このように，2の補数表現による演算では，結果が範囲におさまるよう注意しなければなりません．このために利用されるのがオーバーフローフラグOFです．これは符号なし演算では無意味ですが，我々が2の補数表現による符号つき演算を行う場合に備えて，CPUがオーバーフローを知らせてくれるものです．

CPUは、符号つき演算の場合、正負の符号を表すSFの状態(MSBと同じ)と，桁あふれに使用されるキャリーフラグCFの状態を常に調べていて，両者のXOR (exclusive-or：排他的論理和)をとり，それが1のときにOFを立てています．これは，CFとMSBのどちらか一方が1のときに，オーバーフローが起きるからです．

データ長を2バイトにした場合も、表現できる値が，−32768～32767までに拡張され，MSBが16ビット目に変わるだけで，これまでの話はすべて同じです．

4-4-2 加減算命令

それでは、各演算命令について解説していきましょう．

数値演算の基本は加算と減算です．8086で最も簡単な加減算命令は，INC命令とDEC命令でした．前者はオペランドで指定した，セグメント・レジスタとIPを除く，レジスタおよびメモリの内容に1を加算し，後者は1減算します．メモリのアドレッシングはダイレクトモード，インダイレクトモードがあります．この命令の表現は次のようになります．

INC　Reg (CS, IPを除く全レジスタ)
INC　mem (ダイレクト，インダイレクトによるアドレス)

A+B，A−Bという一般的な2つの数の加算，減算を行うための命令は，ADD命令とSUB(Subtract)命令です．

表4-4-1　ADD, SUB命令のオペランド

命令	オペランド	例	
	DEST ， SORC	バイト	ワード
ADD (SUB)	レジスタ，イミディエイトデータ ＊イミディエイトデータのタイプは，レジスタのタイプにあわせられる	ADD AL, 80 SUB BH, 2	ADD AX, 1F00 SUB BX, 40
ADD (SUB)	メモリ，イミディエイトデータ ＊データのタイプは必ずPTR擬似命令で指定する	ADD BY[130], 30 SUB BY[SI], 20 (BY = BYTE PTR)	ADD WO [BX+SI], 20 SUB WO [DI], 4821 (WO = WORD PTR)
ADD (SUB)	レジスタ，レジスタ	ADD AL, AH SUB BL, CL	ADD BX, SI SUB AX, BX
ADD (SUB)	メモリ，レジスタ ＊メモリのデータタイプはレジスタにあわせられる	ADD [BX], AL SUB [1200], BH	ADD [SI], AX SUB [BX+10], SI
ADD (SUB)	レジスタ，メモリ ＊メモリのデータタイプはレジスタにあわせられる	ADD AL, [200] SUB BL, [BX+5]	ADD AX, [300] SUB DX, [BP+SI]

※ADD, SUBの結果により，AF, CF, OF, PF, SF, ZFが影響される
　ADD, SUBの演算結果はすべてDESTに入る

ADD 命令はすでに紹介しましたが，SUB 命令も ADD と同様，

SUB　DEST , SORC

と表記し，DEST から SORC を減算し，結果を DEST に格納します．このとき，ADD，SUB 命令ともに，SORC の内容は変わりません．

ADD，SUB 命令のオペランドの組み合せを表 4-4-1 に示します．

リスト 4-4-1 は ADD 命令を使ってメモリ中の 10 個のバイトデータの総和をとるものです(図 4-4-3)．

リスト 4-4-1　データの総和を求める 1

```
11BC:0100 BE0900       MOV   SI,0009          ┐
11BC:0103 B90A00       MOV   CX,000A          │
11BC:0106 888C0002     MOV   [SI+0200],CL     │ 0～9をメモリにセットするルーチン
11BC:010A 4E           DEC   SI               │
11BC:010B E2F9         LOOP  0106             ┘
11BC:010D BE0000       MOV   SI,0000
11BC:0110 B90A00       MOV   CX,000A ───────── データ数10個
11BC:0113 33C0         XOR   AX,AX ─────────── AXを0にする命令(後述)
11BC:0115 02840002     ADD   AL,[SI+0200] ──── ALに[SI+200]番地の値を加える
11BC:0119 46           INC   SI
11BC:011A E2F9         LOOP  0115 ──────────── 10回繰り返す
11BC:011C A20A02       MOV   [020A],AL ─────── 答えを20A番地に格納
11BC:011F 90           NOP
```

```
-g=100,11f

AX=0037  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=000A  DI=005C
DS=11BC  ES=11BC  SS=11BC  CS=11BC  IP=011F   NV UP DI PL NZ NA PE NC
11BC:011F 90            NOP
-d200,20f
11BC:0200  01 02 03 04 05 06 07 08-09 0A 37 00 00 00 00 00   ..........7.....
-                    加算されるデータ         └加算結果
```

図 4-4-3　リスト 4-4-1 の実行結果

ADD，SUB 命令単独で計算できる数値は，1 バイト，または 2 バイトデータです．

ADD 命令ではバイト＋バイト，ワード＋バイト（イミディエイトデータ)，ワード＋ワードの 3 つの組合せが 1 命令で行えます．

この中で注意することは，メモリとイミディエイトデータの加算です．SORC または DEST 部にレジスタがあるときは，そのデータタイプ(バイトかワードか)はそのレジスタサイズにより決まってしまいますが，メモリとイミディエイトデータの場合，それがバイトであるかワードであるかわかりません．このようなときは，PTR (Pointer) 演算子を使って，データサイズを指定します．たとえば，メモリの 200H 番地のバイトデータと 35H を加えるには次のようにします．

```
ADD  BYTE  PTR [200] , 35
```

または,

```
ADD [200] , BYTE  PTR  35
```

またワードでは,

```
ADD  WORD  PTR [200] , 3535
```

または,

```
ADD [200] , WORD  PTR  3535
```

(BYTE PTR, WORD PTR は省略して BY, WO とすることもできます)

ここで ADD 命令を使ってメモリ中のバイトデータ (符号なし) を加算して総和を求めるプログラムを作ってみます. これですと, 例えば 130+129 (82H+81H) を行った場合は, 2 つのデータを加えただけで 255 をこえてしまい, 結果は正しく得られません. これをオーバーフロー (桁あふれ) といい, このときは, キャリーフラグ CF がセットされます. そこで, データは 1 バイト型でも, 結果はワードになるように, ワード型の加算を行うようにします. こうすれば, 65535 以内の結果なら正しく求めることができます. このプログラムをリスト4-4-2 に示します (図4-4-4).

リスト 4-4-2 データの総和を求める 2

```
11BC:0100 B90500        MOV     CX,0005 ——— データ数
11BC:0103 BB0002        MOV     BX,0200 ——— 加算されるデータのある先頭アドレス
11BC:0106 BE0000        MOV     SI,0000
11BC:0109 BA0000        MOV     DX,0000
11BC:010C 8A00          MOV     AL,[BX+SI] ┐
11BC:010E B400          MOV     AH,00      ┘ バイトデータをワードに拡張
11BC:0110 03D0          ADD     DX,AX ——— ワードデータの加算
11BC:0112 46            INC     SI
11BC:0113 49            DEC     CX
11BC:0114 75F6          JNZ     010C
11BC:0116 89160502      MOV     [0205],DX ——— 答えを205H番地に格納
11BC:011A 90            NOP
```

```
-g=100,11a

AX=00E0  BX=0200  CX=0000  DX=03C0  SP=FFEE  BP=0000  SI=0005  DI=0000
DS=11BC  ES=11BC  SS=11BC  CS=11BC  IP=011A   NV UP DI PL ZR NA PE NC
11BC:011A 90            NOP
-d200,20f
11BC:0200  [A0 B0 C0 D0 E0] [C0 03] 00-00 00 00 00 00 00 00 00    00P`@..........
-             加算されるデータ   加算結果(03C0H)
```

図4-4-4 リスト4-4-2の実行結果

プログラムは，カウンタに使うためのCXに加算するデータ数5を，BXにデータ領域のベース・アドレスをセットします．結果はDXレジスタに累積するため，最初は0にします．データはまずALにBX+SIのインダイレクトモードで転送します．続いてAHを0にすると，結果的にALの1バイトデータは，AX中の2バイトデータとなります．それから，ADD命令でDXと加算し，DXに結果を格納します．その後ポインタSIとカウンタCXをデクリメントし，JNZによりCXが0なら終了し，そうでなければデータの転送に戻り，加算を繰り返します．

ところで，今は1バイトのデータが符号なしデータであるとして計算しましたが，これが2の補数表現による符号つきデータとした場合は，バイトをワードに変更する時に気をつけなくてはなりません．なぜなら，2の補数表現では負の数はMSBを1としており，たとえば，2バイトデータでの−1の表現は00FFHではなくFFFFHとなるのです．したがって，1バイトの負数を2バイトにするには，上位バイトを，0にするのではなく，すべてのビットを立てる(FFHにする)必要があります．

このような動作を1命令で行うのが，CBW（Convert-Byte-to-Word）です（図4-4-5）.

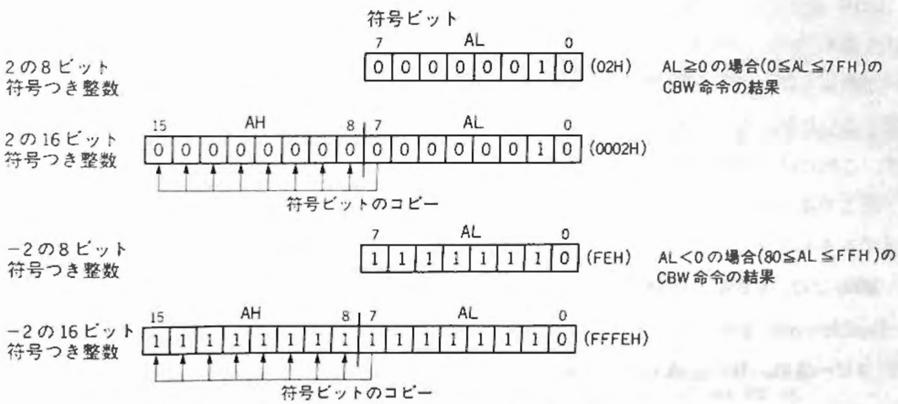


図4-4-5 符号つきバイトデータの16ビット拡張(CBW)

これを使って先程の例のデータを2バイトの2の補数表現データにした場合のプログラムをリスト4-4-3に示します．変ったのは，ALに0を入れていたところをCBW命令に変えた点のみです(図4-4-6)．

リスト 4-4-3 符号つきデータの加算

```
11BC:0100 B90500      MOV   CX,0005
11BC:0103 BB0002      MOV   BX,0200 ———— データのある先頭アドレス
11BC:0106 BE0000      MOV   SI,0000
11BC:0109 BA0000      MOV   DX,0000
11BC:010C 8A00        MOV   AL,[BX+SI]
11BC:010E 98          CBW ———————————— 符号つきバイトデータのワードへの拡張
```

```
11BC:010F 03D0          ADD     DX,AX ──────── ワードデータの加算
11BC:0111 46            INC     SI
11BC:0112 49            DEC     CX
11BC:0113 75F7          JNZ     010C
11BC:0115 89160502      MOV     [0205],DX ──── 答えを205H番地に格納
11BC:0119 90            NOP
```

```
-g=100,119

AX=FFC0  BX=0200  CX=0000  DX=FFB0  SP=FFEE  BP=0000  SI=0005  DI=0000
DS=11BC  ES=11BC  SS=11BC  CS=11BC  IP=0119   NV UP DI PL ZR NA PE CY
11BC:0119 90            NOP
-d200,20f
11BC:0200  10 20 40 80 C0 80 FF 00-00 00 00 00 00 00 00 00   . @.@..........
-           加算されるデータ       加算結果(FFB0H)
       16+32+64+(−128)+(−64)          ‖
                                     −80
```

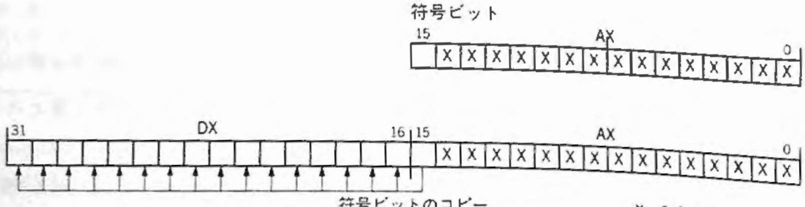
図4-4-6　リスト4-4-3の実行結果

CBW 命令には，このようにオペランドはなく，必ず AL のデータが AX のワードデータに変換されます．

このように，バイト同士，ワード同士の加算で，結果がワード内におさまるものはそのままで行えましたが，もっと大きなデータを計算するためには，さらにデータ長を増やさなくてはならず，これには1回の命令ではすみません．

そこで4バイト（ダブルワード）のデータを加算することを例に，大きな数の加減について考えてみましょう．

8086には，CBW と同様にワードデータをダブルワードに拡張する CWD（Convert-Word-to-Doubleword）命令があります．これを使うと図4-4-7のように AX の最上位ビットが DX 全体にコピーされ，DX と AX で2の補数表示のダブルワード・データになります．



例）AX≧0(0≦AX≦7FFFH)
AX＝07F0H → DX+AX＝000007F0H
AX<0(8000H≦AX≦FFFFH)
AX＝80FFH → DX+AX ＝FFFF80FFH

図4-4-7　符号つきワードデータの32ビット拡張(CWD)

これを行うプログラムを考えてみましょう.

まず，ワードデータ7000HをAXに入れ，これをCWDでダブルワードにします．次にDX,AXをメモリに格納し，ダブルワードデータAとします．さらにC000Hをダブルワードに変換し，データBとしてメモリに納めます．これらのデータの領域の先頭アドレスをBXにして，そのBXでデータを指して計算しメモリCに格納します．このプログラムを**リスト4-4-4**に示します（図4-4-8）.

リスト 4-4-4 ダブルワードデータの加算

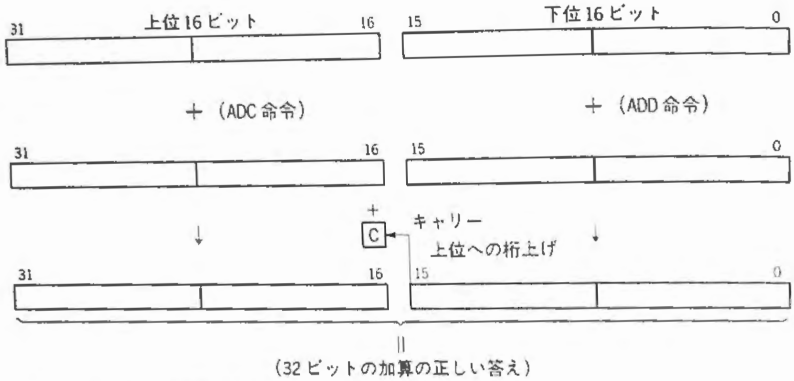
```
11BC:0100 BB0002     MOV    BX,0200 ――― データの先頭アドレス
11BC:0103 99         CWD    ――― AXの値をダブルワードデータにし，上位をDX，下位をAXに格納
11BC:0104 8907       MOV    [BX],AX ――― 下位を[BX]，[BX+1]番地に転送
11BC:0106 895702     MOV    [BX+02],DX ――― 上位を[BX+2]，[BX+3]番地に転送
11BC:0109 8BC1       MOV    AX,CX      ┐
11BC:010B 99         CWD               ┘ CXの値をダブルワードデータに変更
11BC:010C 894704     MOV    [BX+04],AX ――― 下位を[BX+4]，[BX+5]番地に転送
11BC:010F 895706     MOV    [BX+06],DX ――― 上位を[BX+6]，[BX+7]番地に転送
11BC:0112 8B07       MOV    AX,[BX]    ┐
11BC:0114 034704     ADD    AX,[BX+04] ┘ 下位同士の加算
11BC:0117 894708     MOV    [BX+08],AX ――― 結果を[BX+8]，[BX+9]番地に転送
11BC:011A 8B4702     MOV    AX,[BX+02] ┐
11BC:011D 134706     ADC    AX,[BX+06] ┘ 上位同士をキャリーを含めて加算
11BC:0120 89470A     MOV    [BX+0A],AX ――― 結果を[BX+AH]，[BX+BH]番地に転送
11BC:0123 90         NOP
```

```
-rax
AX 0000
:abcd
-rcx
CX 0024
:1234
-g=100,123

AX=FFFF  BX=0200  CX=1234  DX=0000  SP=FFEE  BP=0000  SI=0005  DI=0000
DS=11BC  ES=11BC  SS=11BC  CS=11BC  IP=0123   NV UP DI NG NZ NA PE NC
11BC:0123 90            NOP
-d200,20f
11BC:0200  CD AB FF FF  34 12 00 00  01 BE FF FF  00 00 00 00   M+..4....>.....
-
           AXのダブルワード表現   CXのダブルワード表現   加算結果
           (FFFFABCDH)          (00001224H)          (FFFFBE01H)
```

図4-4-8 リスト4-4-4の実行結果

まずAの下位ワードをAXに転送し，これにインダイレクトモードでBの下位ワードを加え，この結果をCの下位へ格納します．次にAの上位ワードをAXに転送し，これにBの上位ワードを加えますが，このときはADC（Addition-with-Carry）命令を使います．これは，加算の際にキャリーフラグもいっしょに加えるための命令で，これを使えば前の下位ワードの加算により桁あふれが生じ，キャリーが立った場合に，上位ワードの加算に1が加えられるため，正しい結果が得られるのです（**図4-4-9**）．こうして加えた結果をCの上位に格納して演算は終了します.



※ダブルワードデータの加算は1命令で行えないため，上位，下位に分けて行い，
下位はADD命令，上位はADC命令を使う

図4-4-9　ADCの意味

もしADCの代りにADDを使っていたら，キャリーが立った場合，結果は真の値より65536だけ小さいものになってしまうでしょう．プログラムのADCの前に2つのMOV命令がありますが，MOV命令はフラグを変化させないので，問題ありません．

減算命令では，DESTよりSORCの値が大きい場合は，上の桁から桁下がり（繰り下がり）が生じます．これは加算の場合のキャリーにあたるものでボロー（Borrow）といい，MSBへの桁下がりはCFに現れます．減算も加算同様にバイト同士，ワード同士は，範囲内であるなら1命令で正しい結果が得られますが，それ以上ではやはり2つ以上の命令を組み合せなければなりません．

加算と同じように，4バイトデータの減算を行ってみましょう．プログラムを**リスト4-4-5**に，実行結果を**図4-4-10**に示します．ここではADD命令がSUB命令に変わり，ADC命令がSBB（Subtract-with-Borrow）となっています．このSBB命令は，減算の際にDESTからSORCの値を引き，さらにキャリーフラグの値をひいて結果をDESTに格納するものです．この様子を**図4-4-11**に示します．

リスト 4-4-5　ダブルワードデータの減算

```
11BC:0100 BB0002        MOV     BX,0200        ┐
11BC:0103 99            CWD                    │
11BC:0104 8907          MOV     [BX],AX        │
11BC:0106 895702        MOV     [BX+02],DX     │
11BC:0109 8BC1          MOV     AX,CX          │ 加算と同じ
11BC:010B 99            CWD                    │
11BC:010C 894704        MOV     [BX+04],AX     │
11BC:010F 895706        MOV     [BX+06],DX     ┘
11BC:0112 8B07          MOV     AX,[BX]        ┐
11BC:0114 2B4704        SUB     AX,[BX+04]     ┘ 下位同士を減算
11BC:0117 894708        MOV     [BX+08],AX
11BC:011A 8B4702        MOV     AX,[BX+02]     ┐
11BC:011D 1B4706        SBB     AX,[BX+06]     ┘ 上位同士をキャリーを含めて減算
11BC:0120 89470A        MOV     [BX+0A],AX
11BC:0123 90            NOP
```

```
-rax
AX 0000
:abcd
-rcx
CX 0024
:1234
-g=100,123

AX=FFFF  BX=0200  CX=1234  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=11BC  ES=11BC  SS=11BC  CS=11BC  IP=0123   NV UP DI NG NZ NA PE NC
11BC:0123 90            NOP
-d200,20f
11BC:0200  CD AB FF FF  34 12 00 00-99 99 FF FF  00 00 00 00    M+..4...........
```

ABCDHのダブルワード表現（FFFFABCDH）　1234Hのダブルワード表現（00001234H）　減算結果（FFFF9999H）

図4-4-10　リスト4-4-5の実行結果

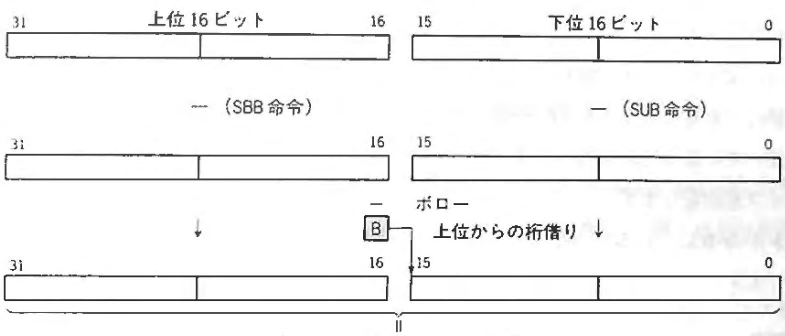


図4-4-11 SBBの意味

4-4-3　乗除算命令

乗除算は高速計算や多量のデータ処理を行う際に，なくてはならない演算です．一昔前のCPUには乗除算命令がなかったため，加減算や後述の論理演算を組み合せて実現していたので，処理速度が大変に遅かったのです．

8086の乗算命令には，MUL（Multiply-accumulator-by-register-or-memory; unsigned）とIMUL（Integer-Multiply-accumulator-by-register-or-memory; signed）があります．

MUL命令は符号なし整数の乗算命令で，バイト同士の乗算およびワード同士の乗算が可能です．バイト同士の乗算はALにデータをセットしてからMUL命令を行います．

```
MOV  AL, data8
MUL  Reg/mem
```

すると結果はAXのワードデータとなります．Reg/memには1バイトメモリ，またはレジスタを指定します．直接の値（イミディエイトデータ）は指定できないことに注意してください．

また，ワード同士の場合は，AX にワードデータをセットし，MUL を行うと，結果は DX：AX に 32 ビットで格納されます．

このとき，上位半分（バイト同士のときは AH，ワード同士のときは DX）が 0 の場合は CF，OF がクリアされて，その他の場合はセットされます．

一方，IMUL 命令は，符号つき整数の乗算用命令ですが，実行は MUL 命令と同じであり，結果の上半分が 0 の場合と，すべてのビットが 1 の場合，つまり上位が下位の符号拡張になっている場合に CF，OF がリセットされ，その他の場合にはセットされる点だけが異なっています．

これらのフラグは，MUL（符号なし乗算），IMUL（符号つき乗算）のどちらの場合も，結果がバイトデータ同士の乗算ならバイトデータに，ワード同士の乗算ならワードデータ内にというように，もとのデータ長内でおさまったかどうかを示しており，もしそうなら下位だけを結果とすることができます．

これは，同じデータ長の場合でしたが，バイトデータにワードデータを乗算するときは，加減算と同様に，符号つきなら CBW で符号拡張を行い，符号なしなら上位バイトを 0 にして，データ長を長い方に合せてから行います．またメモリの指定では，必ず PTR を使ってメモリ中のデータのタイプを指定します．

リスト4-4-6に，いろいろなタイプのデータの乗算を行うプログラムを，図4-4-12に実行結果を示します．

リスト 4-4-6　各種データの乗算例

```
11BC:0100 B380          MOV     BL,80
11BC:0102 B005          MOV     AL,05          ; ① バイトデータ同士の乗算
11BC:0104 F6E3          MUL     BL             ;    結果はAXに入る
11BC:0106 A30002        MOV     [0200],AX      ; 結果の下位，上位を200H，201Hに格納
11BC:0109 B005          MOV     AL,05          ; ② バイトデータ同士(BX=80H)の
11BC:010B F6EB          IMUL    BL             ;    符号つき乗算，結果はAXに入る
11BC:010D A30202        MOV     [0202],AX      ; 結果の下位，上位を202H，203Hに格納
11BC:0110 BB0004        MOV     BX,0400
11BC:0113 B80180        MOV     AX,8001        ; ③ ワードデータ同士の乗算，結果の
11BC:0116 F7E3          MUL     BX             ;    下位がAXに，上位がDXに入る
11BC:0118 A30402        MOV     [0204],AX      ; 下位を204Hに上位を206Hに格納
11BC:011B 89160602      MOV     [0206],DX
11BC:011F B80180        MOV     AX,8001        ; ④ ワードデータの符号つき乗算
11BC:0122 F7EB          IMUL    BX
11BC:0124 A30802        MOV     [0208],AX      ; 結果の下位を208Hに，上位を20AHに格納
11BC:0127 89160A02      MOV     [020A],DX
11BC:012B B081          MOV     AL,81          ; ⑤ バイトデータとワードデータ
11BC:012D B400          MOV     AH,00          ;    (BX=400H)の乗算，データ長を
11BC:012F F7E3          MUL     BX             ;    そろえてから行う
11BC:0131 A30C02        MOV     [020C],AX      ; 結果の格納
11BC:0134 89160E02      MOV     [020E],DX
11BC:0138 B081          MOV     AL,81          ; ⑥ バイトデータとワードデータの符
11BC:013A 98            CBW                    ;    号つき乗算，データ合せにCBW
11BC:013B F7EB          IMUL    BX             ;    を使用
11BC:013D A31002        MOV     [0210],AX      ; 結果の格納
11BC:0140 89161202      MOV     [0212],DX
11BC:0144 BB3412        MOV     BX,1234
```

```
11BC:0147 891E1402        MOV     [0214],BX
11BC:014B B81000          MOV     AX,0010              ⑦ ┐メモリ中のデータ(1234H)との乗
11BC:014E F7261402        MUL     WORD PTR [0214]     ┘算. データ長を明記する
11BC:0152 A31602          MOV     [0216],AX           ┐
11BC:0155 89161802        MOV     [0218],DX           ┘結果の格納
11BC:0159 90              NOP
```

```
-g=100,159

AX=2340  BX=1234  CX=021A  DX=0001  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=11BC  ES=11BC  SS=11BC  CS=11BC  IP=0159   DV UP DI PL NZ NA PO CY
11BC:0159 90             NOP
-          ①の結果 ②の結果    ③の結果       ④の結果        ⑤の結果
-d200,21f   280H    FD80H    2000400H      FE000400H      20400H
11BC:0200  80 02   80 FD   00 04 00 02-00 04 00 FE   00 04 02 00    ...)........~....
11BC:0210  00 04 FE FF   34 12   40 23-01 00   00 00 00 00 00 00    ..~.4.@#........
-            ⑥の結果     メモリ中の   ⑦の結果
            FFFE0400H    データ      12340H
```

図4-4-12　リスト4-4-6の実行結果

除算命令は，DIV（Division; unsigned）命令とIDIV（Integer-Division; signed）命令があります．これは，乗算命令と同じく，DIV命令は符号なし整数での除算に，IDIV命令は符号つき整数での除算に用います．

DIV，IDIV命令ともに，オペランドがバイトの場合は，AXにセットした16ビットの値をバイト・オペランド値で割り，商をALに，余りをAHに格納します．

また，オペランドがワードの場合は，DX：AXにセットした32ビットデータをワード･オペランドの値で割り，商をAX，余りをDXに格納します．

オペランドには，乗算命令同様，レジスタとメモリしか許されません．

この結果，商の値がオーバーフローをおこす場合（たとえば0での除算）には商と余りは不定となり，この時はタイプ0のソフトウェア割込み（後述）が発生します．

この場合を除けば，演算結果は正しくなります．

DIVとIDIVのちがいは，DIVは商が整数でない場合には整数になるように切り捨てを行い，IDIVでは整数部以外の商を切り捨て，余りはオペランドと同じ符号にして格納する点だけです．

除算の例として，リスト4-4-7に1000Hまでの素数を求めるプログラムを示します（図4-4-13）．実行時間は約5秒です．

リスト 4-4-7　素数を求める

```
11BC:0100 BB0002          MOV     BX,0200 ―――――― データエリアの先頭
11BC:0103 BF0200          MOV     DI,0002 ―――――― 最小の素数
11BC:0106 C7070200        MOV     WORD PTR [BX],0002
11BC:010A B90300          MOV     CX,0003
11BC:010D 81F90010        CMP     CX,1000     ┐
11BC:0111 7722            JA      0135        ┘終了判定
```

```
11BC:0113 33F6          XOR     SI,SI
11BC:0115 8BC1          MOV     AX,CX
11BC:0117 33D2          XOR     DX,DX
11BC:0119 F730          DIV     WORD PTR [BX+SI]
11BC:011B 83FA00        CMP     DX,+00
11BC:011E 7502          JNZ     0122
11BC:0120 EB10          JMP     0132
11BC:0122 8BC7          MOV     AX,DI
11BC:0124 48            DEC     AX
11BC:0125 48            DEC     AX
11BC:0126 3BF0          CMP     SI,AX
11BC:0128 7404          JZ      012E
11BC:012A 46            INC     SI
11BC:012B 46            INC     SI
11BC:012C EBE7          JMP     0115
11BC:012E 8909          MOV     [BX+DI],CX
11BC:0130 47            INC     DI
11BC:0131 47            INC     DI
11BC:0132 41            INC     CX
11BC:0133 EBD8          JMP     010D
11BC:0135 90            NOP
```

（11BC:0113〜11BC:012E の注記）その数より小さい素数で順次割っていき，それまでに見つかった最大の素数でも割り切れなければ，あらたな素数としてデータエリアに格納する

```
-g=100,135

AX=0800  BX=0200  CX=1001  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0468
DS=11BC  ES=11BC  SS=11BC  CS=11BC  IP=0135   NV UP DI PL NZ NA PO NC
11BC:0135 90            NOP
-d200,66f
11BC:0200  02 00 03 00 05 00 07 00-0B 00 0D 00 11 00 13 00   ................
11BC:0210  17 00 1D 00 1F 00 25 00-29 00 2B 00 2F 00 35 00   ......%.).+./.5.
11BC:0220  3B 00 3D 00 43 00 47 00-49 00 4F 00 53 00 59 00   ;.=.C.G.I.O.S.Y.
11BC:0230  61 00 65 00 67 00 6B 00-6D 00 71 00 7F 00 83 00   a.e.g.k.m.q.....
11BC:0240                          00-9D 00 A3 00 A7 00 AD 00   ..........#.'.-.
11BC                                  00 C7 00 D3 00 DF 00   3.5.?.A.E.G.S._.
           1D 0E 21 0E 27 0E 2F       FB 00 01 01 07 01   c.e.i.o.q.(.....
  BC:0600  59 0E 5D 0E 6B 0E 71 0E-             33 01 37 01   ........
11BC:0610  95 0E 9B 0E B1 0E B7 0E-B9 0E
11BC:0620  DB 0E ED 0E EF 0E F9 0E-07 0F 0B
11BC:0630  25 0F 29 0F 31 0F 43 0F-47 0F 4D 0F 4F 0F         %.).1.C.G.M.O.S.
11BC:0640  59 0F 5B 0F 67 0F 6B 0F-7F 0F 95 0F A1 0F A3 0F   Y.[.g.k.....!.#.
11BC:0650  A7 0F AD 0F B3 0F B5 0F-BB 0F D1 0F D3 0F D9 0F   '.-.3.5.;.Q.S.Y.
11BC:0660  E9 0F EF 0F FB 0F FD 0F-00 00 00 00 00 00 00 00   i.o.(.).........
-
```

（11BC:0200 行の注記）素数が2バイトで格納されている

図4-4-13　リスト4-4-7の実行結果

4-4-4　BCD演算とASCII補正

これまで説明した数値演算命令は，すべて2進数（16進数）に対するものでした．これは，コンピュータが2進数ですべての情報を表すようにできているためです．しかし，2進数は我々人間が使うには不便このうえないもので，そのため2進数を4ビットずつに区切って16進数とし，少しでも読み易くしようという試みがなされたわけです．

同じようにして，人間になじみの深い10進数を表そうというのが，2進化10進数(BCD: Binary-Coded-Decimal)です．これは，4ビットで表せる0～FHのうち0～9までだけを使用して10進数の1桁を表す方法で，例えば，28を2進および16進で表すと，

0001 1100=1CH

ですが，BCD コードでは，

0010 1000=28

と表すのです．

しかしながら，コンピュータはこの BCD コードを直接計算するようにはできていません．

例えば，18+25 を行うとすると，そのまま BCD コードをコンピュータに加算させれば，

```
    18        0001 1000
+) 25     +) 0010 0101
    43 ≠      0011 1101 = 3DH
```

のように答えは違ってしまいます．また 10 進数には D などはありません．

よく考えればあたりまえのこと，繰り上がりの桁が異なっているのです．

8086 には，これを正しく補正するための命令が用意されています．これらは 10 進補正命令といい，BCD で使用しない A～F までの値が出た場合に，自動的に 6 を加えて正しい BCD コードにするものです．これを先程の加算に適用すれば，次のようになります．

```
    0011 1101        3D
+) 0000 0110    +) 06
    0100 0011        43
```

これは 10 進補正命令のひとつである DAA（Decimal Adjust for Addition）命令の結果です．

これをプログラムで書くと，

```
MOV   AL, 18
ADD   AL, 25
DAA
```

となります．

このように DAA 命令は加算命令の直後におき，AL のみに作用する命令です．

ところで，2 進化 10 進数には，4 ビットを 1 桁とするものの他に 8 ビットを 1 桁とするものもあり，前者をパックド 10 進数，後者をアンパックド 10 進数といいます．パックド 10 進数の加算補正命令が DAA であったのに対し，アンパックド 10 進数に対する加算補正命令には AAA(Ascii-Adjust-for-Addition)を使用します．

ASCII 補正という名は，数を文字として表している ASCII コードからきています．ASCII コードでは 0～9 までの数値を 30～39H の 16 進 1 バイトデータで表します．

一方，アンパックド 10 進数も，1 バイトで 0～9 までの値を表しており，各データに 30H を加

えるだけでASCIIコードに変換することができるのです．

10進補正命令には，この他にパックド，アンパックド10進数の減算補正のためのDAS（Decimal-Adjust-for-Subtraction），AAS（ASCII-Adjust-for-Subtraction），アンパックド10進数同士の乗算後の補正を行うAAM（ASCII-Adjust-for-Multiply），そしてアンパックド10進数同士の除算前に被除数（レジスタにセットする値）の補正を行うAAD（ASCII-Adjust-for-Division）があります．

これらの命令の実行結果でキャリーやボローが生じた場合，キャリーフラグがセットされるので，多数桁の10進演算を行う時は，1桁の計算の後に必ず補正を行う必要があります．

これらの10進補正命令について**表4-4-2**にまとめておきます．また**リスト4-4-8**は，4桁（2バイト）のBCDコードの加減算を行うプログラムです（**図4-4-14**）．

表4-4-2　10進補正命令

演算名	10進補正命令名	
	パックド(4ビット／1桁)	アンパックド(8ビット／1桁)
加　算	DAA	AAA
減　算	DAS	AAS
乗　算	――――	AAM
除　算	――――	AAD *

※これらはALレジスタのみに働く命令である
結果によりフラグは変化する

＊AADは，除算前に被除数に対して行う

リスト 4-4-8　2進10進補正

```
11BC:0100 BB0002        MOV     BX,0200      ― データ領域の先頭
11BC:0103 B88046        MOV     AX,4680      ┐
11BC:0106 8907          MOV     [BX],AX      │ 演算されるデータをメモリに格納
11BC:0108 B87935        MOV     AX,3579      │
11BC:010B 894702        MOV     [BX+02],AX   ┘
11BC:010E 8A07          MOV     AL,[BX]      ┐ 各データの下位バイトをレジスタに転送
11BC:0110 8A6702        MOV     AH,[BX+02]   ┘
11BC:0113 02C4          ADD     AL,AH        ┐ 加算後10進補正を行う
11BC:0115 27            DAA                  ┘
11BC:0116 884704        MOV     [BX+04],AL   ― メモリに結果を格納
11BC:0119 8A4701        MOV     AL,[BX+01]   ┐ 各データの上位バイトをレジスタに転送
11BC:011C 8A6703        MOV     AH,[BX+03]   ┘
11BC:011F 12C4          ADC     AL,AH        ┐ キャリーを含めて加算後10進補正を行う
11BC:0121 27            DAA                  ┘
11BC:0122 884705        MOV     [BX+05],AL   ― メモリに結果を格納
11BC:0125 8A07          MOV     AL,[BX]      ┐ 各データの下位バイトを再度レジスタに転送
11BC:0127 8A6702        MOV     AH,[BX+02]   ┘
11BC:012A 2AC4          SUB     AL,AH        ┐ 減算後10進補正を行う
11BC:012C 2F            DAS                  ┘
11BC:012D 884706        MOV     [BX+06],AL   ― メモリに結果を格納
11BC:0130 8A4701        MOV     AL,[BX+01]   ┐ 各データの上位バイトをレジスタに転送
11BC:0133 8A6703        MOV     AH,[BX+03]   ┘
```

```
11BC:0136 1AC4          SBB      AL,AH       ]キャリーも含めて減算後10進補正を行う
11BC:0138 2F            DAS                  ]
11BC:0139 884707        MOV      [BX+07],AL ――メモリに結果を格納
11BC:013C 90            NOP
```

```
-g=100,13c

AX=3511  BX=0200  CX=003D  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0468
DS=11BC  ES=11BC  SS=11BC  CS=11BC  IP=013C   NV UP DI PL NZ NA PE NC
11BC:013C 90            NOP
-
-d200,20f
11BC:0200  80 46  79 35  59 82  01 11-00 00 00 00 00 00 00 00   .Fy5Y...........
           データA データB A,Bの和 A,Bの差
           4680    3579    8259    1101
```

図4-4-14　リスト4-4-8の実行結果

4-4-5 論理演算

これまでの算術演算とは別に，8086には論理演算用の命令があります．これらの命令は，レジスタやメモリ上のデータの各ビットに対して処理を行うことができ，これを使えば，さらに複雑なデータ形式を作ったり，I/Oデバイスの細かいコントロールが行えます．

論理演算の主要なものは，否定をとるNOT命令，論理積をとるAND命令，論理和をとるOR命令，排他的論理和をとるXOR命令があります．

○ NOT命令

NOT命令は，1バイトまたは1ワードのデータの各ビットの1，0を反転する働きをします．オペランドにはレジスタ及びメモリが指定でき，メモリの指定では，そのタイプをPTRによって指定します．例えば，次のようにAXの値の否定をとるには，

```
NOT   AX
```

1バイトのメモリの内容の否定は，

```
NOT   BYTE   PTR [200]
```

のようにします．もし，AXまたは200H番地のデータが3FHだった場合，結果は次のようになります．

```
      3FH=0011 1111
NOT   3FH=1100 0000=C0H
```

またある数を，NOT命令によって各ビットの1，0を反転してから1を加えると，その数にマイナスをつけた数の2の補数が求められます．これを1命令で行うのがNEG(Negative)命令です．

○ AND 命令

論理積は，AND 命令により求められます．これは，オペランドを 2 つ指定し，その 2 つのデータの対応する各ビットについて 1 つずつ独立に論理積をとります．この命令は次のように使います．

```
MOV  AL , F8
AND  AL , 18
```

このとき，AL データ F8H とイミディエイトデータ 18H の各ビットが AND され，結果は

```
        F8H =1111 1000
AND)    18H =0001 1000
---------------------
        18H =0001 1000
```

となります．

ここでわかるように，AND 命令によって，DEST と SORC を 2 進で表現した場合の双方ともに，1 の立っているビットを調べることができます．つまり，同じ場所が 1 であれば結果も 1 となり，そうでなければ 0 になるからです．この結果，ひとつも同じ所がない場合は ZF がセットされます．

ところで，AND の結果は DEST に入るため，オペランドの内容が変更されてしまいます．これに対して，以前に述べた TEST 命令は同様に AND 操作を行いますが，結果はフラグのみに表され，内容は変化しません．

AND では DEST が AL，AX の場合は省略して AND，SORC と書くことができます．

また，メモリとイミディエイトデータの演算には PTR によるタイプの指定が必要です．

```
AND  WORD  PTR [200] , 3
```

○ OR 命令

論理和は OR 命令を用います．AX レジスタのデータとイミディエイトデータとの論理和は，次のように書きます．

```
MOV  AX , F800
OR   AX , 4210
```

結果は，

```
      F800H     1111 1000 0000 0000
OR)   4210H     0100 0010 0001 0000
-----------------------------------
      FA10H     1111 1010 0001 0000
```

となり，このように 2 つのオペランドの各ビットの，論理和がとられます．OR 命令は，DEST のデータの任意のビットを立てたいときに使います．

○ XOR 命令

排他的論理和は，XOR (exclusive-OR) 命令で行います．これは2つのデータの同じビットが異なっていれば1に，同じであれば0にするものです．よく使う例として，レジスタの値を0にするのにレジスタ自身のデータの XOR をとります．

```
XOR  AX , AX
```

以上の命令のまとめを図 4-4-15 に示します．

命令	DEST，SORC	例
AND OR XOR	レジスタ，イミディエイトデータ メモリ，イミディエイトデータ レジスタ，レジスタ メモリ，レジスタ レジスタ，メモリ	AND AL, 80　または　AND, 80 OR BYTE PTR [SI], 42 XOR AX, BX AND [BX+SI], AX OR CX, [BP]
NOT NEG	レジスタ メモリ	NOT AL NEG WORD PTR [BX+SI]

※これらの演算結果はすべて DEST に入る

図4-4-15　論理演算命令のまとめ

これらの命令の使用例としてグラフィック V-RAM 上に論理演算によって作ったパターンを描くプログラムを紹介しました（リスト 4-4-9，図 4-4-16）．

リスト 4-4-9　V-RAMにパターンを描く

```
11BD:0100 B80000    MOV   AX,0000  ── グラフィックメモリ・フィルルーチンの引数
11BD:0103 E84300    CALL  0149     ── 全画面のクリア
11BD:0106 E87E00    CALL  0187     ── テキスト画面表示停止
11BD:0109 E86300    CALL  016F     ── グラフィック画面表示開始
11BD:010C 06        PUSH  ES
11BD:010D B8FFFF    MOV   AX,FFFF  ┐
11BD:0110 BA00A8    MOV   DX,A800  │ A8000H～AFFFFHまでをFFFFHでフィル
11BD:0113 E84A00    CALL  0160     ┘
11BD:0116 B800A8    MOV   AX,A800
11BD:0119 8EC0      MOV   ES,AX    ── ESにグラフィックメモリのセグメント値セット
11BD:011B BF1400    MOV   DI,0014  ── 繰り返し回数セット(20回)
11BD:011E BE0000    MOV   SI,0000
11BD:0121 B8F0F0    MOV   AX,F0F0  ── XORパターン1(この値を変えると模様が変わる)
11BD:0124 B90040    MOV   CX,4000  ┐
11BD:0127 26        ES:            │
11BD:0128 3104      XOR   [SI],AX  │ グラフィックメモリとのXOR
11BD:012A 46        INC   SI       │
11BD:012B 46        INC   SI       │
11BD:012C E2F9      LOOP  0127     ┘
11BD:012E E86600    CALL  0197     ── 表示ウエイト
11BD:0131 B80F0F    MOV   AX,0F0F  ── XORパターン2(この値を変えると模様が変わる)
11BD:0134 B90040    MOV   CX,4000  ┐
11BD:0137 26        ES:            │
```

（メインルーチン：11BD:0100～，ループ：11BD:011E～）

```
11BD:0138 3104          XOR     [SI],AX      ┐
11BD:013A 46            INC     SI           │ グラフィックメモリとのXOR
11BD:013B 46            INC     SI           │
11BD:013C E2F9          LOOP    0137         ┘
11BD:013E 4F            DEC     DI ―――――――― 繰り返しカウンタのデクリメント
11BD:013F 75DD          JNZ     011E
11BD:0141 07            POP     ES
11BD:0142 E83A00        CALL    017F ―――――― グラフィック画面表示停止
11BD:0145 E84700        CALL    018F ―――――― テキスト画面表示開始
11BD:0148 90            NOP ――――――――――――― プログラム終了(ブレークポイントを置くアドレス)
11BD:0149 06            PUSH    ES           全グラフィックメモリのフィルルーチン
11BD:014A 50            PUSH    AX             (入力：AX＝フィルパターン)
11BD:014B BA00A8        MOV     DX,A800      ┐
11BD:014E E80F00        CALL    0160         │
11BD:0151 BA00B0        MOV     DX,B000      │ グラフィックメモリのセグメント値
11BD:0154 E80900        CALL    0160         │ のセット
11BD:0157 BA00B8        MOV     DX,B800      │
11BD:015A E80300        CALL    0160         ┘
11BD:015D 58            POP     AX
11BD:015E 07            POP     ES
11BD:015F C3            RET
11BC:0160 57            PUSH    DI           グラフィックメモリのフィル(1色分)
11BC:0161 51            PUSH    CX             (引数：AX＝フィルパターン
11BC:0162 8EC2          MOV     ES,DX                 DX＝グラフィックメモリのセグメント)
11BC:0164 BF0000        MOV     DI,0000
11BC:0167 B90040        MOV     CX,4000
11BC:016A F3            REPZ
11BC:016B AB            STOSW
11BC:016C 59            POP     CX
11BC:016D 5F            POP     DI
11BC:016E C3            RET
11BD:016F 51            PUSH    CX           グラフィック画面表示開始ルーチン
11BD:0170 50            PUSH    AX           640×400，カラーモードに設定
11BD:0171 FA            CLI
11BD:0172 B5C0          MOV     CH,C0
11BD:0174 B442          MOV     AH,42
11BD:0176 CD18          INT     18 ―――――――― BIOSコール
11BD:0178 B440          MOV     AH,40
11BD:017A CD18          INT     18
11BD:017C 58            POP     AX
11BD:017D 59            POP     CX
11BD:017E C3            RET
11BD:017F 50            PUSH    AX           グラフィック画面表示停止ルーチン
11BD:0180 FA            CLI
11BD:0181 B441          MOV     AH,41
11BD:0183 CD18          INT     18
11BD:0185 58            POP     AX
11BD:0186 C3            RET
11BD:0187 50            PUSH    AX           テキスト画面表示停止ルーチン
11BD:0188 FA            CLI
11BD:0189 B40D          MOV     AH,0D
11BD:018B CD18          INT     18
11BD:018D 58            POP     AX
11BD:018E C3            RET
11BD:018F 50            PUSH    AX           テキスト画面表示開始ルーチン
11BD:0190 FA            CLI
11BD:0191 B40C          MOV     AH,0C
11BD:0193 CD18          INT     18
11BD:0195 58            POP     AX
11BD:0196 C3            RET
```

サブルーチン

```
11BD:0197 51            PUSH    CX          時間待ちルーチン
11BD:0198 B9FFFF        MOV     CX,FFFF
11BD:019B E2FE          LOOP    019B
11BD:019D 59            POP     CX
11BD:019E C3            RET
```

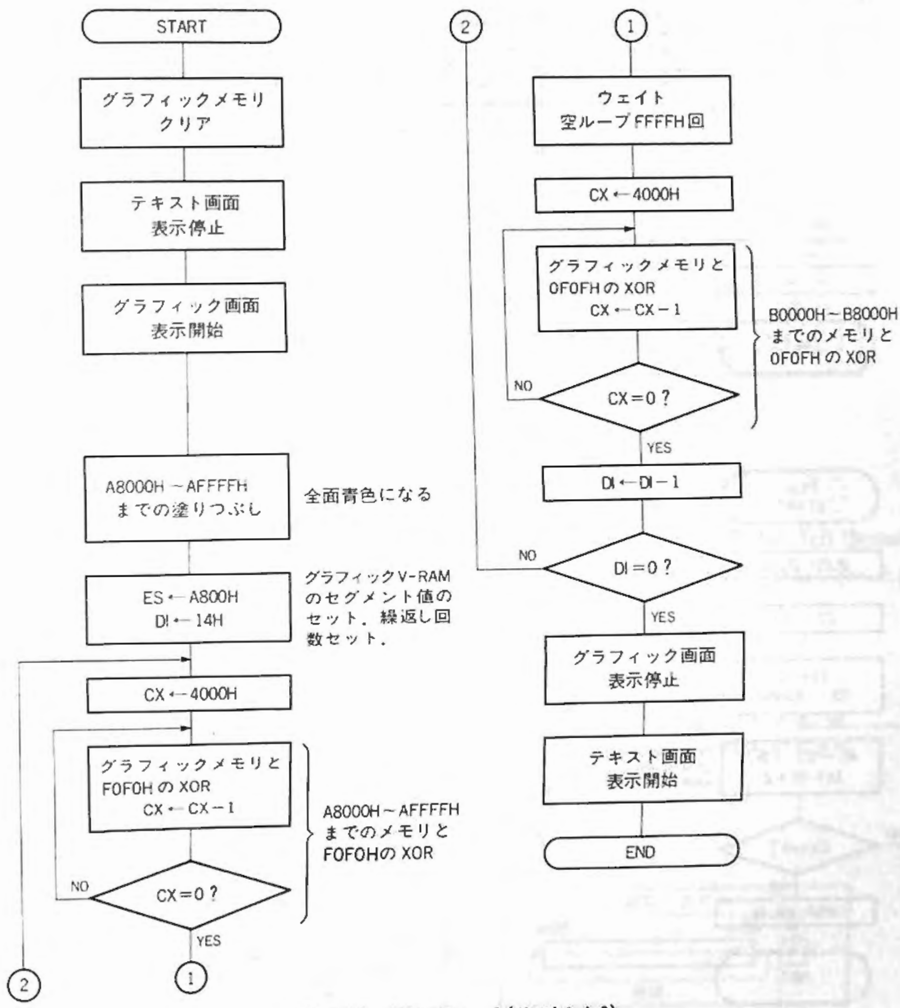


図4-4-16 フローチャート(リスト4-4-9)

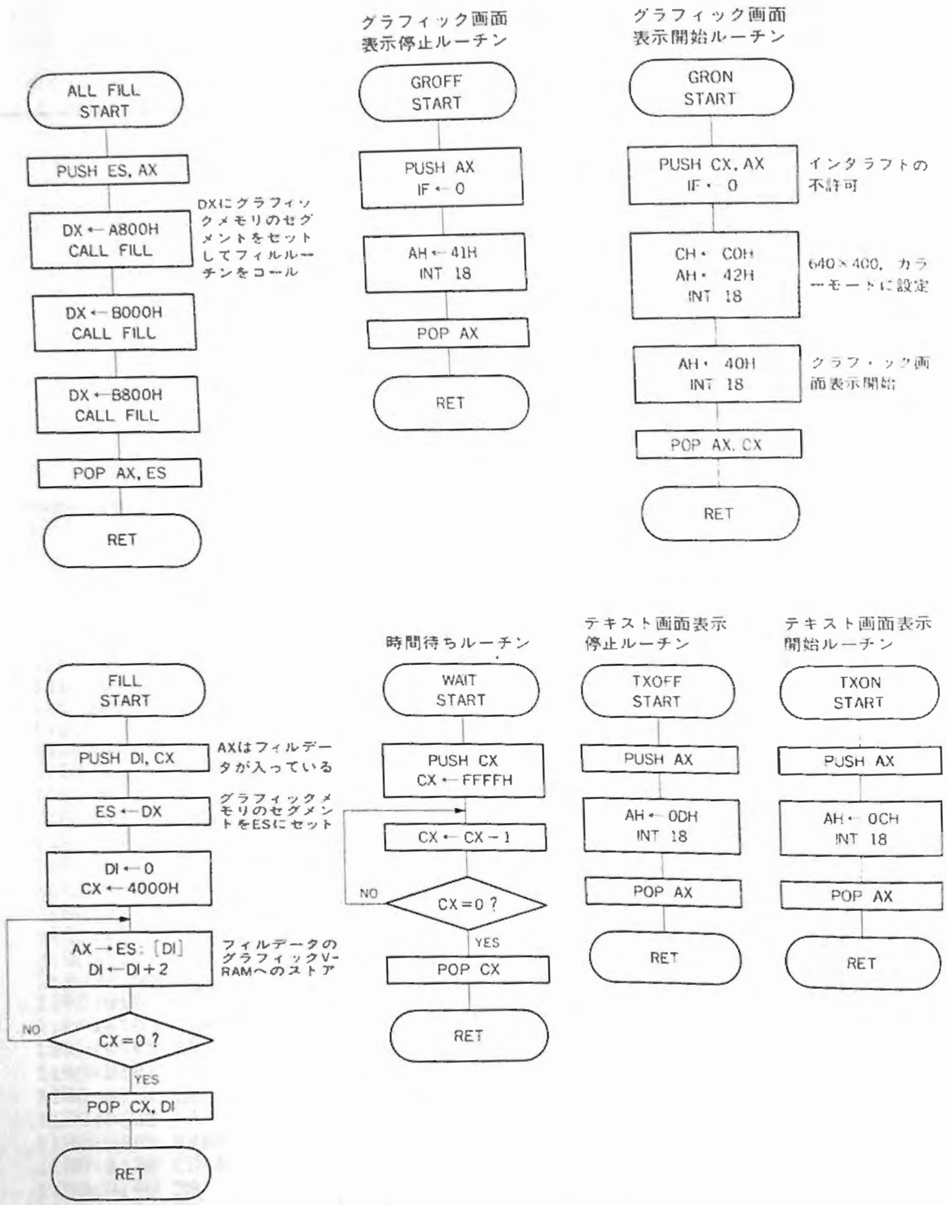
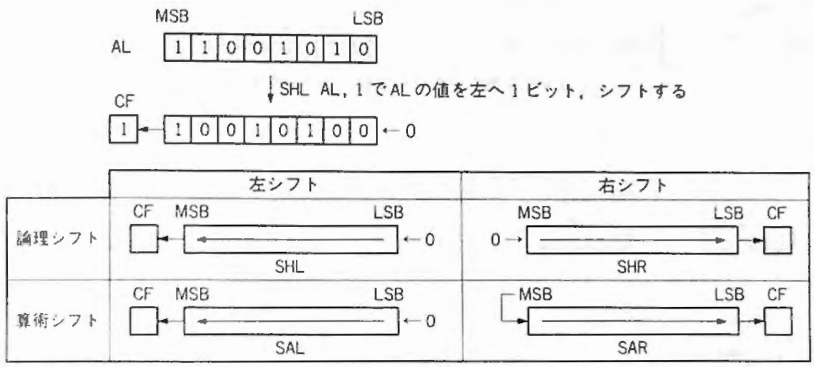


図4-4-16　フローチャート（続き）

論理演算の中には，レジスタやメモリ上のデータを，ビット列とみなしてシフトや回転を行う命令があります．

○シフト命令

シフトを行う命令には SHL(SAL) (Shift-logical-Left; Shift-Arithmetic-Left), SAR(Shift-Arithmetic-Right), SHR (Shift-logical-Right) があります．これらは，データを右または左に1つずつシフトしていき，シフトによりはみ出したビットを CF に入れます．またシフトにより空いた側には SHL(SAL)，SHR では 0 が入り，SAR では MSB の値がコピーされます(図4-4-17)．

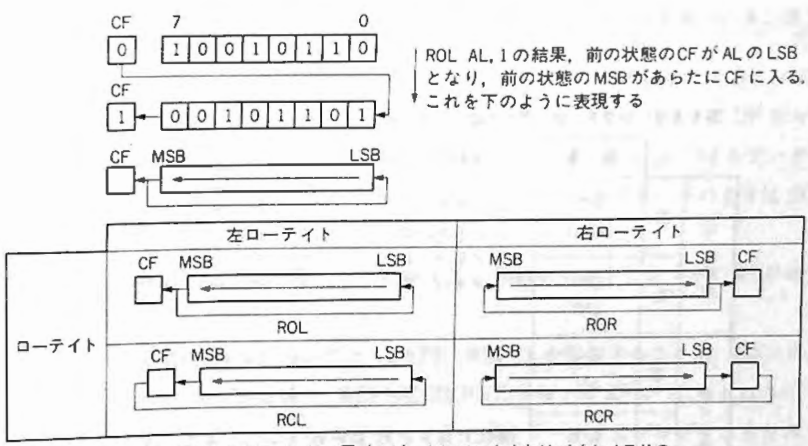


※SHL, SAL は同じ命令である

図4-4-17　シフト命令の動作

○ローテイト命令

ローテイト命令である ROL (Rotate left)，ROR (Rotate Right)，RCL (Rotate left through Carry)，RCR (Rotate right through Carry) は，図4-4-18に示すようにデータの左右への回転を行います．それぞれの回転では必ず CF にデータが入ります．



※ROL, ROR では，データがもとに戻るには8回（8ビットレジスタまたはバイトメモリの場合）回転させればよいが，RCL, RCR では，CF を通るため9回，回転する必要がある

図4-4-18　ローテイト命令の動作

これらの命令を使うと，データを2倍,4倍にしたり1/2倍，1/4倍にすることが簡単に行えますし,MUL命令でも扱えない桁の多い数の乗算を行うこともできます．この例を図4-4-19に示しました．

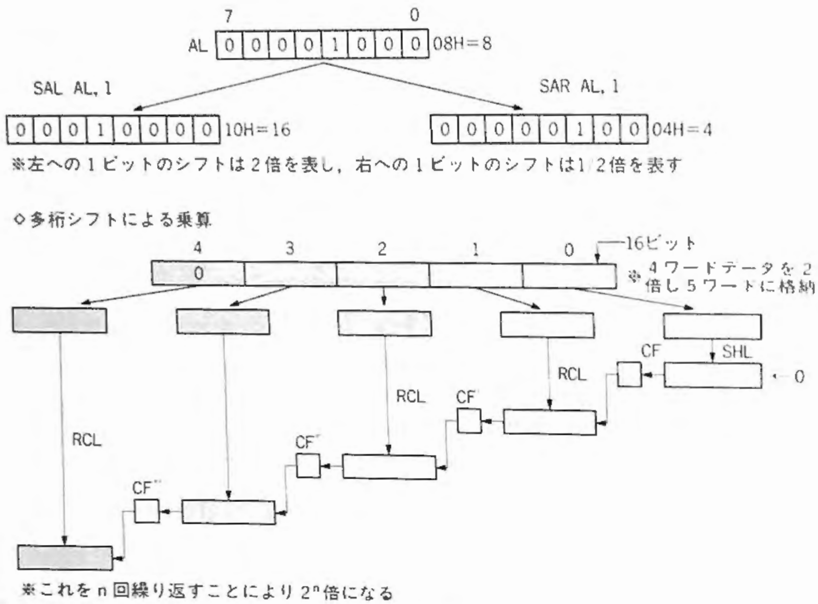


図4-4-19　シフト，ローテイトによる乗除算

これらの命令は，オペランドのDEST部にシフト，ローテイトするレジスタやメモリを，またSORC部にその回数を指定します．シフト，ローテイト命令についてのまとめを表4-4-3に示します．

表4-4-3　シフト，ローテイト命令のまとめ

		命令	DEST, SORC	例
シフト命令	左	SHL (SAL)	レジスタ，1 レジスタ，CL メモリ，1 メモリ，CL	SHL AL, 1 ROL AX, CL RCL BY [100], 1 SAR WO [BX], CL
	右	SHR SAR		
ローテイト命令	左	ROL RCL		
	右	ROR RCR	※CLにはシフト，ローテイトの回数を指定する	

4-5　データの検索とソーティング

4-5-1　ストリング命令によるデータの検索

データ処理では，いくつかのデータの中から，ある特定のデータと同じものを捜し出すことをよく行います．ここではこのようなデータの検索（サーチ）をマシン語で行う方法を説明します．8086には，1バイトや1ワードのデータの検索に便利な，SCAS命令が用意されています．これを利用して，図4-5-1に示すような，メモリ上のデータ領域（200～2FFH）の中から55Hというデータを捜し出して，その数を数えるプログラムを考えてみましょう．

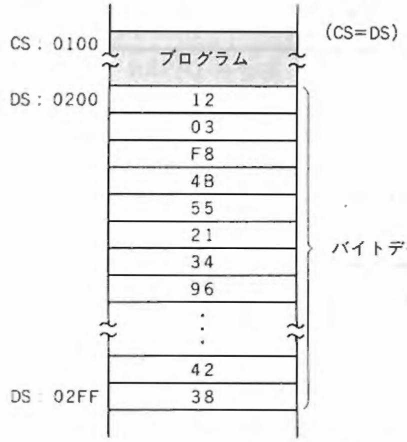


図4-5-1　データサーチ・プログラムのメモリマップ

SCASとは，Scan byte(word)Stringのことで，これもストリング命令の1つです．その動作は，ES内のDIの値で示されるメモリの内容（バイトまたはワード）をALまたはAXレジスタの値から減算し，フラグ類を変化させます．そして，DIの値をディレクションフラグDFが0ならインクリメントし，1ならデクリメントします．インクリメント，デクリメントは，バイトデータなら1回，ワードデータなら2回行います．ニーモニックは，バイトデータのサーチのときはSCASB，ワードデータのサーチのときはSCASWを使用します．

ストリング命令の繰返しは，MOVS命令と同様，CXに回数をセットしたうえでREP命令で行います．

SCAS命令およびCMPS命令については，REPE（REPZ）を使用することで，CX=0となる前に，ZFが0になればループから抜け，REPNE(REPNZ)を使えばZFが1になればループから抜ける，ということができます．この命令を使ってデータをサーチするプログラムをリスト4-5-1に示します（図4-5-2）．

リスト 4-5-1 サーチルーチン

```
11BC:0100 8CD8       MOV    AX,DS     ]DSとESを等しくする
11BC:0102 8EC0       MOV    ES,AX     ]
11BC:0104 FC         CLD
11BC:0105 BF0002     MOV    DI,0200 ── データエリアの先頭アドレス
11BC:0108 B055       MOV    AL,55
11BC:010A B300       MOV    BL,00
11BC:010C B90001     MOV    CX,0100
11BC:010F F2         REPNZ ──────────── CXが0かゼロフラグがセットするまで
                                         次のストリング命令を実行する
11BC:0110 AE         SCASB ──────────── DIの示すアドレスのデータとAL(55H)を比較する
11BC:0111 83F900     CMP    CX,+00 ──── 終了判定("+"は入力しない)
11BC:0114 7404       JZ     011A
11BC:0116 FEC3       INC    BL ──────── 55HがあればBLをインクリメント
11BC:0118 EBF5       JMP    010F
11BC:011A 90         NOP
```

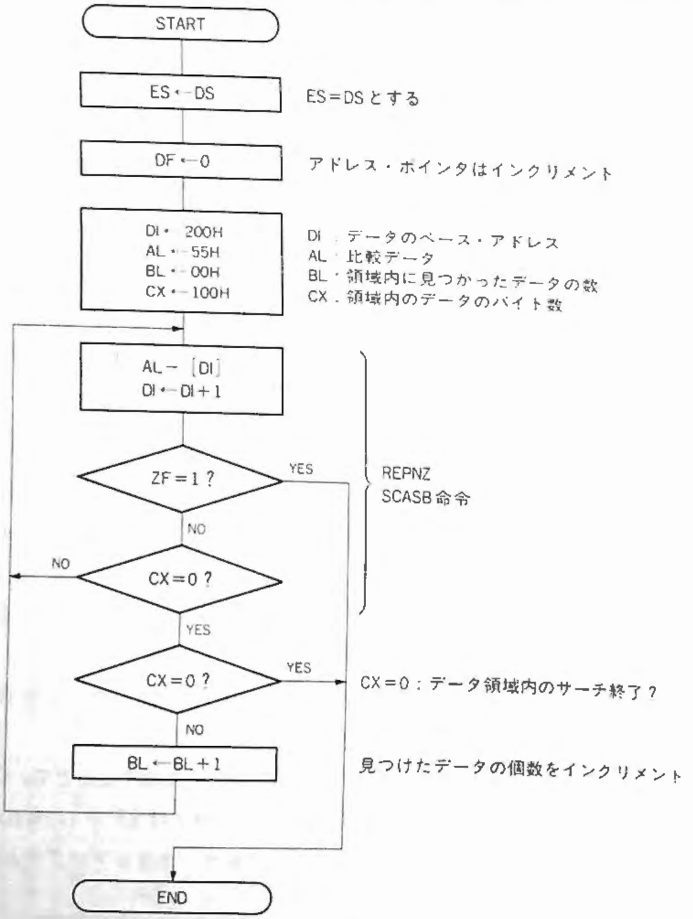


図4-5-2 フローチャート(リスト4-5-1)

まず REPNZ 命令を使用する準備として CLD 命令（Clear Direction Flag）でディレクションフラグを 0 にし，サーチする方向をインクリメントにします．そして DI にサーチを開始するアドレス 200H，CX にサーチをするバイト数256 をセットします．また BL には 55H と同じデータの数を入れるため，あらかじめ 0 を入れます．そしてREPNZ,SCASBによりサーチが行われ，CX は減っていきます．その中で 55H が見つかると，ZF=1 となるので，ループから抜け，次の命令で，CX が 0 かどうか調べます．もしそうならば全領域のサーチを完了したことになり，そうでなければ BL を 1 ふやし，再び REPNZ 命令へジャンプします．このとき，CX と DI は前回リピート動作を抜けたときの値を保持しているので，重なることなく残りの領域のサーチが続けられます．

このプログラムの実行の際には，E コマンドで，DS：200H からのメモリに 55H を含めたデータを 256 バイト分用意しておいて下さい．

プログラムをトレースすると，REPNZ 命令は，図 4-5-3 のように次の SCASB 命令まで一度に実行されます．また IP の値は変化しないことがわかります．これは他の REP 命令でも同じことがいえます．

```
-t=100

AX=11BD  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=11BD  ES=11BD  SS=11BD  CS=11BD  IP=0102   NV UP DI PL NZ NA PE NC
11BD:0102 8EC0          MOV     ES,AX
-t7

AX=11BD  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=11BD  ES=11BD  SS=11BD  CS=11BD  IP=0105   NV UP DI PL NZ NA PE NC
11BD:0105 BF0002        MOV     DI,0200

AX=11BD  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0200
DS=11BD  ES=11BD  SS=11BD  CS=11BD  IP=0108   NV UP DI PL NZ NA PE NC
11BD:0108 B055          MOV     AL,55

AX=1155  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0200
DS=11BD  ES=11BD  SS=11BD  CS=11BD  IP=010A   NV UP DI PL NZ NA PE NC
11BD:010A B300          MOV     BL,00

AX=1155  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0200
DS=11BD  ES=11BD  SS=11BD  CS=11BD  IP=010C   NV UP DI PL NZ NA PE NC
11BD:010C B90001        MOV     CX,0100

AX=1155  BX=0000  CX=0100  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0200
DS=11BD  ES=11BD  SS=11BD  CS=11BD  IP=010F   NV UP DI PL NZ NA PE NC
11BD:010F F2            REPNZ
11BD:0110 AE            SCASB

AX=1155  BX=0000  CX=00FF  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0201
DS=11BD  ES=11BD  SS=11BD  CS=11BD  IP=010F   NV UP DI PL NZ NA PE NC
11BD:010F F2            REPNZ
11BD:0110 AE            SCASB

AX=1155  BX=0000  CX=00FE  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0202
DS=11BD  ES=11BD  SS=11BD  CS=11BD  IP=010F   NV UP DI PL NZ NA PE NC
11BD:010F F2            REPNZ
11BD:0110 AE            SCASB
-
```

命令が繰り返されている

図4-5-3　サーチプログラムのトレース

4-5-2 バブルソートによるデータの並べ替え

データの検索とともによく行うのが，データのソートです．これはデータが複数あるときに，それらを小さい順，または大きい順に並べ替えることで，そのアルゴリズムは多くのものが研究されています．ここでは，バブルソートと呼ばれるアルゴリズムで，データを小さい順にソートするプログラムを作ってみます．アルゴリズムの説明のために，図 4-5-4 を見てください．

アルゴリズム

```
FOR I=N TO 1 STEP-1
 FOR J=0 TO I-1
  IF Dj > Dj+1 THEN
   SWAP Dj,Dj+1
NEXT J,I
```

I	J	配列データ D_0	D_1	D_2	D_3	D_4
—	—	4	2	0	1	3
4	0	2	4	0	1	3
4	1	2	0	4	1	3
4	2	2	0	1	4	3
4	3	2	0	1	3	4
3	0	0	2	1	3	4
3	1	0	1	2	3	4
3	2	0	1	2	3	4
2	0	0	1	2	3	4
2	1	0	1	2	3	4
1	0	0	1	2	3	4

(n=4)　実行の順序

図 4-5-4　バブルソートのアルゴリズムとソーティングの様子

図 4-5-4 のように，バブルソートはデータを 1 次元配列中にとり，これを前から 2 つずつデータを比較して，前のデータが後のデータより大きければ，入れ替える作業を行い，これを繰り返すことによって大きなデータが，泡が水中を動くかのように後ろへ移動していきながらソートが行われるので，その名前がついたものです．

これをプログラムしたものをリスト 4-5-2 に示します (図 4-5-5)．

リスト 4-5-2 バブルソートルーチン

```
11BD:0100 BF0001        MOV     DI,0100  ―― データの個数
11BD:0103 BB0002        MOV     BX,0200  ―― データ領域の先頭
11BD:0106 4F            DEC     DI
11BD:0107 83FF01        CMP     DI,+01   ] 終了判定
11BD:010A 721D          JB      0129     ]
11BD:010C 8BCF          MOV     CX,DI
11BD:010E 49            DEC     CX
11BD:010F BE0000        MOV     SI,0000
11BD:0112 3BF1          CMP     SI,CX    ] 1度目の入れ替えの終了判定
11BD:0114 77F0          JA      0106     ]
11BD:0116 8A00          MOV     AL,[BX+SI]
11BD:0118 8A6001        MOV     AH,[BX+SI+01]
```

```
11BD:011B 3AC4          CMP     AL,AH        ┐[BX+SI]番地の値を，その次の
11BD:011D 7607          JBE     0126         │番地の値を比較して，大きければ，
11BD:011F 86C4          XCHG    AH,AL        │入れ替える
11BD:0121 8800          MOV     [BX+SI],AL   │
11BD:0123 886001        MOV     [BX+SI+01],AH┘
11BD:0126 46            INC     SI
11BD:0127 EBE9          JMP     0112──────────無条件分岐
11BD:0129 90            NOP
```

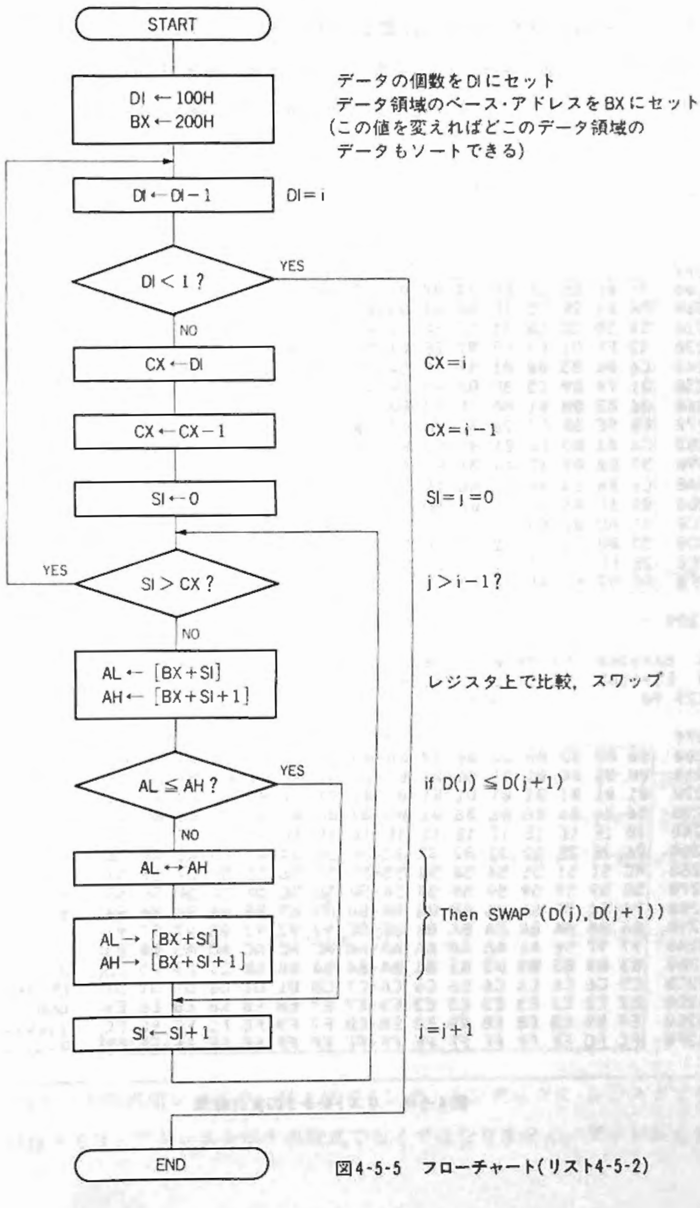


図4-5-5　フローチャート(リスト4-5-2)

プログラムは，符号なしのバイトデータを扱うこととして，BP にデータの先頭アドレス，CX にデータ数をセットします．DI, SI をインデックス i，インデックス j として BP+SI によってデータを指し，D(j)を AL に，D(j+1)を BL に転送して比較し，AL>BL なら XCHG(Exchange) 命令によって入れ替えを行っています．XCHG 命令は，

```
XCHG   DEST , SORC
```

により，DEST と SORC のデータを入れ替えます．

DEST，SORC としてはバイト・レジスタ，ワード・レジスタおよびメモリが指定できますが，そのデータ長は合せなくてはいけません．また XCHG 命令の実行後もフラグ・レジスタの内容は変化しません．

図 4-5-6 にこのプログラムの実行の様子を示します．

```
                              ソート前のデータ
-d200,2ff
11BD:0200  7F 0E 0E 1F FF 24 B4 03-EB 08 B0 03 B4 01 EB 02   .....$4.k.0.4.k.
11BD:0210  B4 01 2E C5 1E 8A 01 89-47 03 5B 07 1F 5D 5F 5A   4..E....G.[..]_Z
11BD:0220  59 58 5E CB 51 57 FC C6-06 83 00 01 8A 1E 54 00   YX^KQW|F.3....T.
11BD:0230  32 FF D1 E3 FF 97 8E 01-5F 59 AA E2 E7 EB D1 FC   2.Qc...._Y*bgkQ|
11BD:0240  C6 06 B3 00 01 BA 1E 54-00 32 FF D1 E3 FF 97 92   F.3....T.2.Qc...
11BD:0250  01 74 09 C5 3E 8A 01 88-45 0D EB B4 EB AB FC C6   .t.E>...E.k4k(|F
11BD:0260  06 B3 00 01 8A 1E 54 00-32 FF D1 E3 FF 97 96 01   .3....T.2.Qc....
11BD:0270  EB 9E 8B F7 26 AC E8 04-00 E2 F9 EB 93 51 56 FC   k..w&,h..byk.QV|
11BD:0280  C6 06 B3 00 01 86 C1 8A-1E 56 00 88 1E AC 01 BB   F.3...A..V....,.;
11BD:0290  57 00 89 1E AA 01 EB 4B-01 EB 68 59 5E 59 C3 FC   W...*.hK.hhY^YC|
11BD:02A0  C6 06 B3 00 01 8A 1E 56-00 88 1E AC 01 BB 57 00   F.3....V...,.;W.
11BD:02B0  89 1E AA 01 C7 06 AD 01-00 00 E8 45 01 E8 44 59   ..*.G.-...hE.hDY
11BD:02C0  A1 AD 01 E9 4C FF 51 57-FC C6 06 83 00 01 8A 1E   !-.iL.QW|F.3....
11BD:02D0  55 00 32 FF D1 E3 FF 97-8E 01 5F 59 AA E2 E7 E9   U.2.Qc...._Y*bgi
11BD:02E0  2E FF FC C6 06 83 00 01-8A 1E 55 00 32 FF D1 E3   ..|F.3....U.2.Qc
11BD:02F0  FF 97 92 01 74 0A C5 3E-8A 01 88 45 0D E9 10 FF   ....t.E>...E.i..
-
-g=100,129

AX=0000  BX=0200  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0001  DI=0000
DS=11BD  ES=11BD  SS=11BD  CS=11BD  IP=0129   NV UP DI NG NZ AC PE CY
11BD:0129 90            NOP
-
                              ソート後のデータ
-d200,2ff
11BD:0200  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
11BD:0210  00 00 00 01 01 01 01 01-01 01 01 01 01 01 01 01   ................
11BD:0220  01 01 01 01 01 01 01 01-01 01 01 02 03 03 03 04   ................
11BD:0230  06 06 06 06 06 06 06 06-07 08 09 0A 0D 0D 0E 0E   ................
11BD:0240  10 1E 1E 1E 1E 1E 1E 1E-1E 1E 1E 1E 1E 1F 1F 24   ...............$
11BD:0250  26 2E 2E 32 32 32 32 32-3E 3E 44 45 45 45 47 4B   &..22222>>DEEEGK
11BD:0260  4C 51 51 51 54 54 54 55-55 56 56 56 57 57 57 57   LQQQTTTUUVVVWWWW
11BD:0270  58 59 59 59 59 59 59 5A-5B 5D 5E 5E 5F 5F 5F 68   XYYYYYYZ[]^^___h
11BD:0280  74 74 7F 81 86 88 88 88-88 89 89 89 8A 8A 8A 8A   tt..............
11BD:0290  8A 8A 8A 8A 8A 8A 8B 8E-8E 92 92 93 96 97 97 97   ................
11BD:02A0  97 97 9E A1 A8 AA AA AA-AA AC AC AC AD AD B0 B3   ...!(****,,,--03
11BD:02B0  B3 B3 B3 B3 B3 B3 B4 B4-B4 B4 BB BB C1 C3 C5 C5   3333334444;;ACEE
11BD:02C0  C5 C6 C6 C6 C6 C6 C6 C6-C7 CB D1 D1 D1 D1 D1 D1   EFFFFFFFGKQQQQQQ
11BD:02D0  E2 E2 E2 E3 E3 E3 E3 E3-E7 E7 E8 E8 E8 E8 E8 E9   bbbccccccghhhhhi
11BD:02E0  E9 E9 EB EB EB EB EB EB-EB F7 F9 FC FC FC FC FC   iikkkkkkkwy|||||
11BD:02F0  FC FC FF FF FF FF FF FF-FF FF FF FF FF FF FF FF   ||..............
-
```

図4-5-6　リスト4-5-2の実行結果

4-6 その他のマシン語命令

4-6-1 入出力命令

第2章で述べましたが，8086は外部の入出力用周辺デバイスを，メモリ空間とは別のI/O空間に置き，I/Oポートとのデータの入出力を専用の入出力命令を用いて行います．

I/Oポートからのデータの入力には，IN命令を使います．ポートの指定は，DXレジスタにI/O空間内のポート番号（0～FFFFH）をセットして行う，レジスタ・アドレッシングモードと，0～FFHのポートを数値で直接指定するイミディエイトモードの2つの方法があり，どちらも，ポート内容は，ALまたはAXレジスタに転送されます．

```
IN   AL , 20
IN   AX , DX
```

同様にI/O空間へのデータの出力にはOUT命令を用います．OUT命令のオペランドはIN命令のDESTとSORC部を入れ換えた形となり，ALまたはAXの内容をI/Oポートへ出力します．

```
OUT   20 , AX
OUT   DX , AL
```

入出力命令の使用例は，第7章で紹介します．

4-6-2 アドレスロード命令

インダイレクト・アドレッシングモードでは，あるメモリ・アドレスの指定を次のように，ベース・レジスタ(BP, BX)インデックス・レジスタ(DI, SI)および8ビット，16ビット定数を組み合せて行うことができました．

```
MOV   AX , [BX+SI+5]
```

これは，メモリ上に多次元の配列変数をとる場合に大変便利なアドレッシング法ですが，このように指定したアドレスが，実際はどのオフセット・アドレスを示しているのかを知りたい場合があります．具体的には，SI＝100H，BX＝40Hのときに，[BX+SI+5]の実際のオフセット値（40H＋100H＋5H＝145H）を求めたい場合です．

8086はこれをLEA（Load-Effective-Address）命令によって行うことができます．これは次のように表現します．

```
LEA   Reg 16 , [mem]
```

DEST部は16ビットの汎用レジスタ，およびポインタ，インデックス・レジスタでなくてはならず，SORC部はメモリ・アドレスを示す表現式でなくてはなりません．アドレスを直接代入す

るには，

```
LEA  AX , [1234]
```

のようにアドレスに角カッコをつけます．

LEA 命令の例として，グラフィック V-RAM 上に，図形を描くプログラムをリスト 4-6-1 に示します（図 4-6-1）．

このプログラム中で LEA 命令は，グラフィック V-RAM 中の座標を，実アドレスに変換することに使用しています．

リスト 4-6-1 LEA命令を用いたV-RAMのアクセス

```
; メインルーチン
11BD:0100 8D1EF501   LEA   BX,[01F5]
11BD:0104 B90800     MOV   CX,0008
11BD:0107 BE0700     MOV   SI,0007
11BD:010A B001       MOV   AL,01
11BD:010C B202       MOV   DL,02
11BD:010E 8800       MOV   [BX+SI],AL
11BD:0110 4E         DEC   SI
11BD:0111 F6E2       MUL   DL
11BD:0113 E2F9       LOOP  010E
      ; (0100～0113) XLAT 命令で使うテーブルを作る
      ; テーブルには，点を打つためのビットパターンを用意する
      ; パターンは下位メモリアドレスから80H，40H，20H，10H，08H，04H，02H，01H である
11BD:0115 E8C500     CALL  01DD        ; グラフィック画面表示開始
11BD:0118 BB0003     MOV   BX,0300     ; BOXデータの先頭アドレス
11BD:011B BF0300     MOV   DI,0003
11BD:011E B90200     MOV   CX,0002
11BD:0121 8A01       MOV   AL,[BX+DI]
11BD:0123 3CFF       CMP   AL,FF
11BD:0125 7423       JZ    014A
      ; (0121～0125) Y2データがFFHのときは終了
11BD:0127 32E4       XOR   AH,AH
11BD:0129 50         PUSH  AX
11BD:012A 4F         DEC   DI
11BD:012B 8A01       MOV   AL,[BX+DI]
11BD:012D B402       MOV   AH,02
11BD:012F F6E4       MUL   AH
11BD:0131 50         PUSH  AX
11BD:0132 4F         DEC   DI
11BD:0133 E2EC       LOOP  0121
      ; 座標データは，300H番地よりX1，Y1，X2，Y2の順で1バイトずつ格納しており，X座標ではこれを2倍した値を，Y座標はそのままを，それぞれ16ビットに拡張してBOXルーチンにスタックに積んで引き渡す．指定するX，Yの範囲は，0≦X1，X2≦FFH，0≦Y1，Y2≦FEHでX1<X2，Y1<Y2 の条件を満たすこと
11BD:0135 8B4F04     MOV   CX,[BX+04]  ; グラフィックメモリのセグメント値のセット
11BD:0138 8BEC       MOV   BP,SP
11BD:013A B001       MOV   AL,01
11BD:013C E80C00     CALL  014B
11BD:013F B90400     MOV   CX,0004
11BD:0142 58         POP   AX
11BD:0143 E2FD       LOOP  0142
      ; (013F～0143) SPの値を戻すためPOPする
11BD:0145 83C306     ADD   BX,+06      ; "+"は入力しないこと
11BD:0148 EB01       JMP   011B
11BD:014A 90         NOP               ; メインルーチンの終了
11BD:014B 06         PUSH  ES          ; BOXルーチンの開始
11BD:014C 53         PUSH  BX
11BD:014D 8B7604     MOV   SI,[BP+04]
11BD:0150 2B7600     SUB   SI,[BP+00]  ; X2−X1の距離を計算
11BD:0153 8B5E00     MOV   BX,[BP+00]
11BD:0156 8D18       LEA   BX,[BX+SI]  ; この演算はADD BX,SIと同じ
11BD:0158 8B5602     MOV   DX,[BP+02]  ; スタック上から引数を受けとる
11BD:015B E82600     CALL  0184
11BD:015E 8B5606     MOV   DX,[BP+06]
      ; Yを一定にしてXを変えて横線を描く
```

```
11BD:0161 E82000       CALL  0184
11BD:0164 4E           DEC   SI
11BD:0165 79EC         JNS   0153                        BOXルーチン (X1,Y1)−(X1,Y1)を対角線とする
11BD:0167 8B7E06       MOV   DI,[BP+06]
11BD:016A 2B7E02       SUB   DI,[BP+02]                  Y2−Y1の距離を計算
11BD:016D 8B5E02       MOV   BX,[BP+02]
11BD:0170 8D11         LEA   DX,[BX+DI]                  LEAにより，DX=BX+DIという演算が1命令で行える
11BD:0172 8B5E00       MOV   BX,[BP+00]
11BD:0175 E80C00       CALL  0184
11BD:0178 8B5E04       MOV   BX,[BP+04]
11BD:017B E80600       CALL  0184                        PSETルーチン
11BD:017E 4F           DEC   DI
11BD:017F 79EC         JNS   016D                        Xを一定にしてYを変えて縦線を描く
11BD:0181 5B           POP   BX
11BD:0182 07           POP   ES
11BD:0183 C3           RET
11BD:0184 50           PUSH  AX                          PSETルーチンの開始
11BD:0185 53           PUSH  BX                          入力 AL：1ドットのON/OFF(1/0)
11BD:0186 51           PUSH  CX                               BX：X座標の値
11BD:0187 52           PUSH  DX                               DX：Y座標の値
11BD:018B 56           PUSH  SI                               CX：グラフィックRAMのセグメント
11BD:0189 57           PUSH  DI
11BD:018A A2FD01       MOV   [01FD],AL
11BD:018D 891EFE01     MOV   [01FE],BX                   ワークエリアに引数をコピーしておく
11BD:0191 B9160002     MOV   [0200],DX
11BD:0195 8EC1         MOV   ES,CX                       セグメントレジスタのセット(CXは引数)
11BD:0197 33D2         XOR   DX,DX
11BD:0199 8BC3         MOV   AX,BX
11BD:019B B90800       MOV   CX,0008                     X/8の演算
11BD:019E F7F1         DIV   CX
11BD:01A0 A30202       MOV   [0202],AX                   X/8(商)
11BD:01A3 88160402     MOV   [0204],DL                   XMOD8(余り)
11BD:01A7 33D2         XOR   DX,DX
11BD:01A9 A10002       MOV   AX,[0200]
11BD:01AC B95000       MOV   CX,0050                     Y*80
11BD:01AF F7E1         MUL   CX                          PSETルーチン (X,Y)のドットセット・リセット
11BD:01B1 8BF0         MOV   SI,AX
11BD:01B3 8B1E0202     MOV   BX,[0202]
11BD:01B7 8D38         LEA   DI,[BX+SI]                  (X,Y)の点のあるVRAMのアドレス
11BD:01B9 26           ES:
11BD:01BA 8A0D         MOV   CL,[DI]                     現在描かれているドットを調べる
11BD:01BC BD1EF501     LEA   BX,[01F5]                   パターンテーブルのアドレスをBXにセット
11BD:01C0 A00402       MOV   AL,[0204]                   ALに何ビット目かを入れ，対応するパターンを得る
11BD:01C3 D7           XLAT
11BD:01C4 803EFD0101   CMP   BYTE PTR [01FD],01          ドットONか?
11BD:01C9 7406         JZ    01D1
11BD:01CB F6D8         NEG   AL                          (X,Y)のドットをOFFにする
11BD:01CD 22C1         AND   AL,CL
11BD:01CF EB02         JMP   01D3
11BD:01D1 0AC1         OR    AL,CL                       ONなのでORをとる
11BD:01D3 26           ES:
11BD:01D4 8805         MOV   [DI],AL                     データの書き込み
11BD:01D6 5F           POP   DI
11BD:01D7 5E           POP   SI
11BD:01D8 5A           POP   DX
11BD:01D9 59           POP   CX
11BD:01DA 5B           POP   BX
11BD:01DB 58           POP   AX
11BD:01DC C3           RET
11BD:01DD 51           PUSH  CX                          グラフィック画面表示開始ルーチン
```

```
11BD:01DE 50             PUSH   AX
11BD:01DF FA             CLI
11BD:01E0 B5C0           MOV    CH,C0
11BD:01E2 B442           MOV    AH,42
11BD:01E4 CD18           INT    18
11BD:01E6 B440           MOV    AH,40
11BD:01E8 CD18           INT    18
11BD:01EA 58             POP    AX
11BD:01EB 59             POP    CX
11BD:01EC C3             RET
11BD:01ED 50             PUSH
11BD:01EE FA             CLI
11BD:01EF B441           MOV    AH,41
11BD:01F1 CD18           INT    18
11BD:01F3 58             POP    AX
11BD:01F4 C3             RET
-d300
```

Annotations on the listing:

- 01DE–01EC: 画面表示開始ルーチン
- MOV CH,C0 / MOV AH,42: 表示モード：640×400，カラーモード
- MOV AH,40: 表示開始
- INT 18 (01E8): BIOSコール
- PUSH (01ED): グラフィック画面表示停止ルーチン
- 01ED–01F4: 画面表示停止ルーチン
- INT 18 (01F1): BIOSコール
- ＊グラフィック画面を消したいときには，このルーチンをコールすればよい

```
-d300,3ff
11BD:0300  00 00 FF FE 00 A8 00 00-FF FE 00 B0 00 00 FF FE   ...~.(...~.0...~
11BD:0310  00 B8 0E 1F FF 24 00 A8-B4 03 FF 24 00 B0 03 B4   .8...$.(4..$.0.4
11BD:0320  24 EB 00 B8 B4 03 EB 08-00 AB 03 B0 08 EB 00 B0   $k.84.k..(.0.k.0
11BD:0330  B0 03 EB 08 00 B8 03 B0-08 B4 00 AB B0 03 B4 81   0.k..8.0.4.(0.4.
11BD:0340  00 B0 03 84 81 EB 00 B8-B4 02 EB 81 00 A8 02 B4   .0.4.k.84.k..(.4
11BD:0350  81 EB 00 B0 B4 01 EB 02-00 B8 01 2E 02 B4 00 A8   .k.04.k..8...4.(
11BD:0360  2E 01 B4 C5 00 B0 01 1E-C5 2E 00 B8 1E 8C 2E C5   ..4E.0..E..8...E
11BD:0370  00 A8 8C 01 C5 1E 00 B0-01 89 1E 8C 00 B8 89 01   .(..E..0.....8..
11BD:0380  8C 47 00 A8 01 03 47 89-00 B0 03 47 89 5B 00 B8   .G.(..G..0.G.[.8
11BD:0390  47 03 5B 07 00 A8 03 1F-07 5B 00 B0 1F 07 5B 5D   G.[..(...[.0..[]
11BD:03A0  00 B8 07 1F 5D 5F 00 A8-1F 5A 5F 5D 00 80 5A 59   .8..]_.(.Z_].0ZY
11BD:03B0  5D 5F 00 B8 59 58 5F 5A-00 A8 58 59 5A 5E 00 B0   ]_.8YX_Z.(XYZ^.0
11BD:03C0  59 58 5E CB 00 88 58 51-CB 5E 00 A8 51 57 5E CB   YX^K.8XQK^.(QW^K
11BD:03D0  00 B0 57 51 CB FC 00 88-51 57 FC C6 00 A8 57 06   .0WQK|.8QW|F.(W.
11BD:03E0  C6 FC 00 B0 06 84 FC C6-00 B8 B4 00 C6 06 00 A8   F|.0.4|F.84.F..(
11BD:03F0  00 01 06 84 00 B0 00 00-00 FF 00 00 00 00 00 00   ...4.0..........
-
```

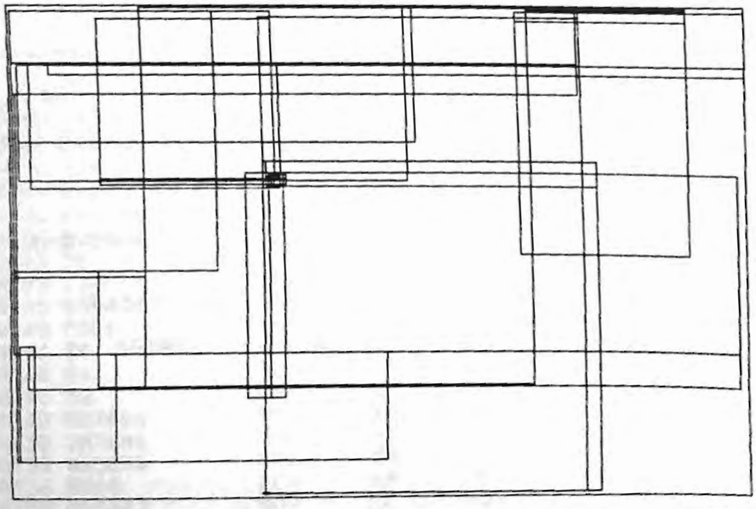


図4-6-1　座標データと出力結果

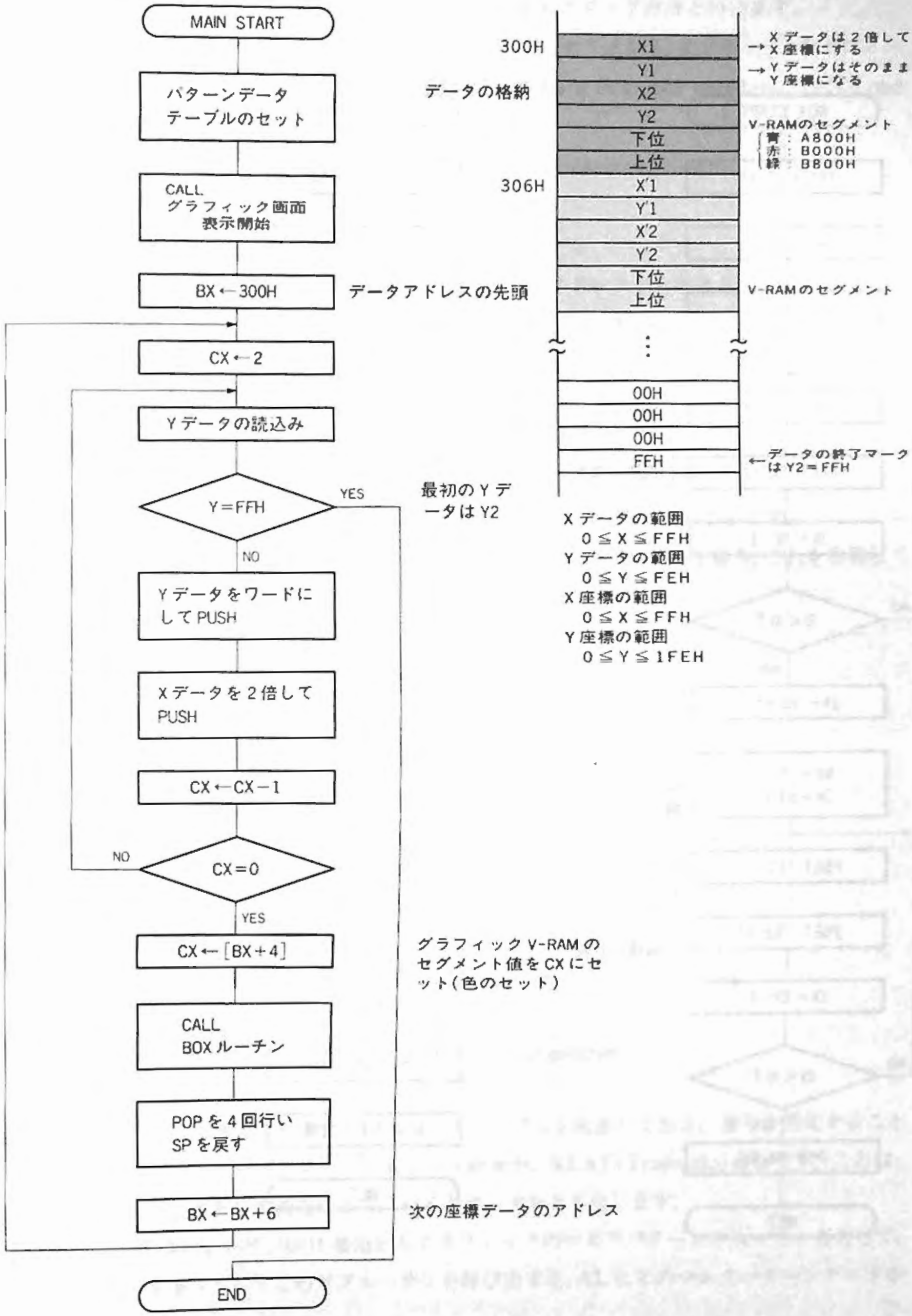


図4-6-2　フローチャート(リスト4-6-1)

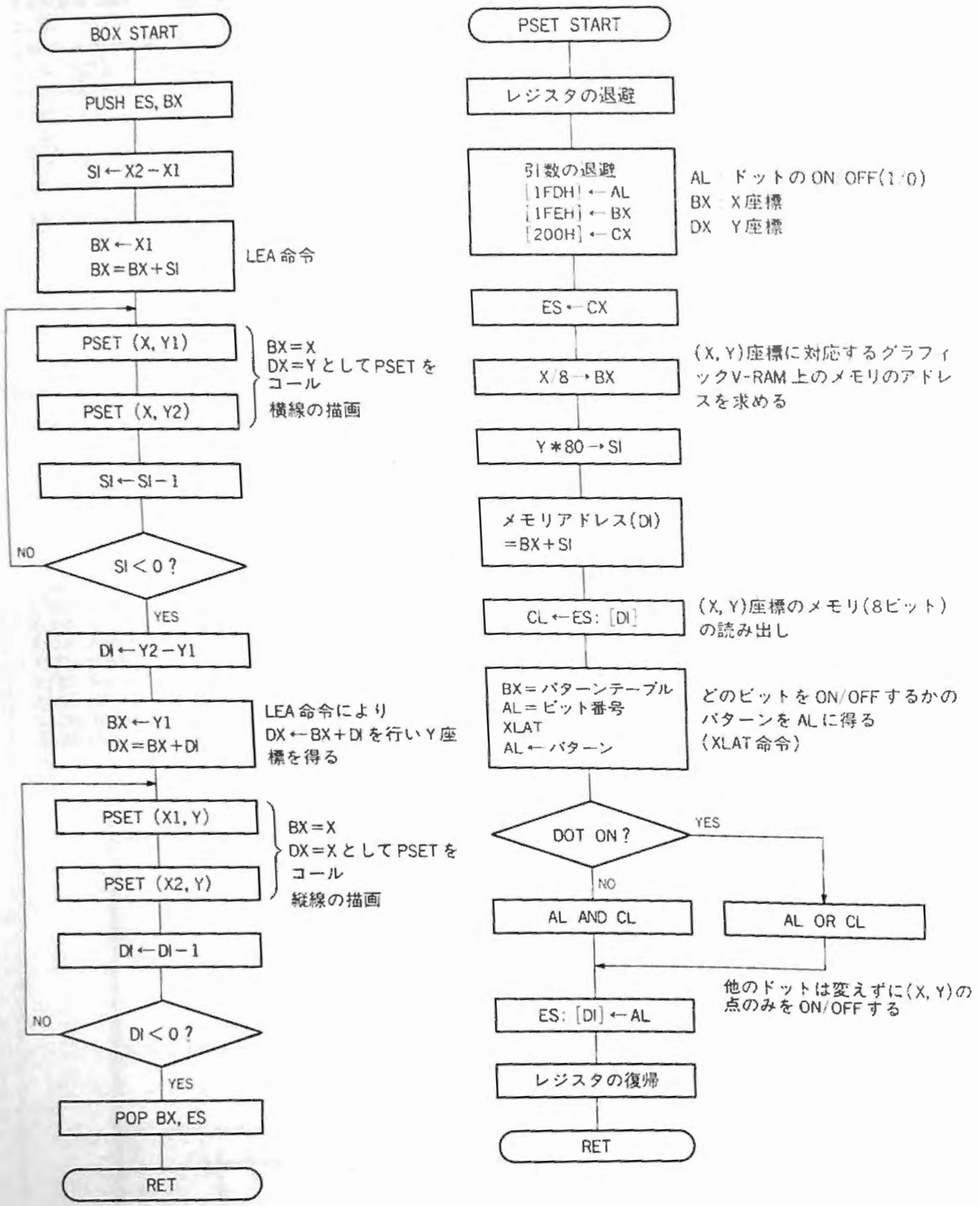


図 4-6-2 フローチャート(続き)

このLEAのようにアドレスの値を求める命令を，アドレスロード命令といいます．

アドレスロード命令には，この他に4バイトのアドレス（セグメント，オフセット）をセグメント・レジスタと16ビットレジスタへ格納するLDS(Load-Data-Segment-register)，LES(Load-Extra-Segment-register）命令があります．

LDS命令は，

```
LDS  Reg 16 , [mem]
```

のように指定することで，SORC部で指定されたメモリ・アドレスから始まる4バイトの絶対アドレスのセグメント・アドレスをDSに，DEST部に指定した汎用16ビット・レジスタ，SI，DI，BX，BPのいずれかへ，オフセット・アドレスを転送します．

また，LES命令も同様に，

```
LES  Reg 16 , [mem]
```

と指定します．LDS命令との違いは，セグメント・アドレスがESに入ることです．

これらの命令は，メモリ中に図4-6-3に示すようなアドレステーブルを置き，これを参照してセグメント外データのアクセスを行うような時に使用できます．

◇右のプログラムは左図のようなアドレステーブルがあるときの利用例である

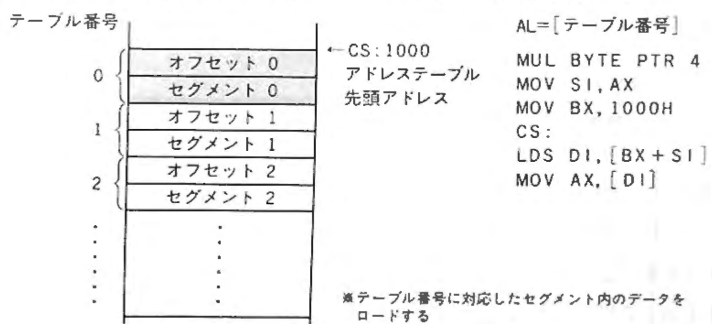


図4-6-3　アドレステーブルの参照の例

アドレステーブルの参照と同じように，データテーブルを用意しておき，番号を指定することにより，対応しているデータをロードするという命令が，XLAT(Translate)命令です．これは，BXとALを用いたアドレス指定により，バイトデータをとり出します．

例えば図4-6-4において，100H番地からグラフィックのマスクパターンが入っているとして，ALにその番号をセットしてこのサブルーチンを呼び出すと，ALにそのマスクパターンデータが得られます．

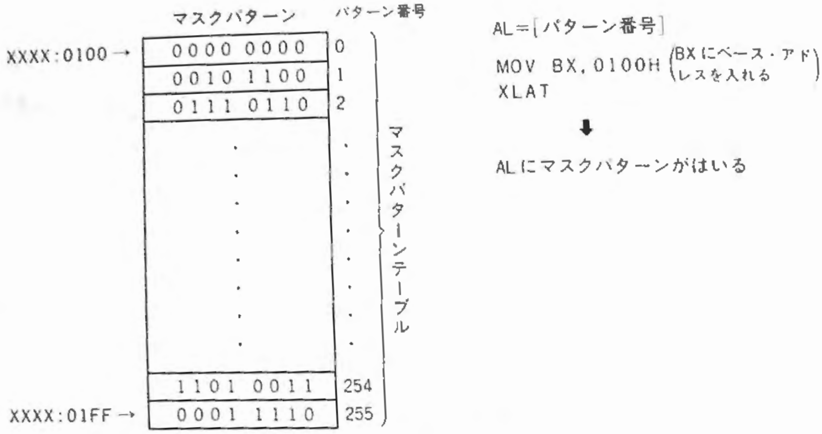


図4-6-4　XLATの例

4-6-3　フラグ制御命令

フラグ・レジスタの値はCMP命令や算術演算命令の実行によって変化することは前に説明しましたが，8086には任意のフラグをセットしたり，クリアしたりするフラグ制御命令があります．これらは，あらかじめフラグの状態を決めておきたい時や，ZF（ゼロフラグ）をセットしてすぐにJZ命令を行うことで，条件分岐命令の条件として使うなどの使い方があります．これを表4-6-1にまとめます．

表4-6-1　フラグ制御命令

命令	意　味
CLC	CF（キャリーフラグ）を0（クリア）にする
CMC	CFを反転する
STC	CFを1（セット）にする
CLD	DF（ディレクションフラグ）をクリアしてストリング命令の実行を増加方向にする
STD	DFをセットしてストリング命令の実行を減少方向にする
CLI	IF（インタラプトフラグ）をクリアして外部割込みを不許可とする
STI	IFをセットして外部割込みを許可する

ところでフラグ・レジスタは，CALL命令や割込みの生じた時に自動的にスタックへPUSHされますが，これをプログラムで積極的に行うための命令として，PUSHF(Push-Flag-register), POPF (Pop-Flag-register) 命令が用意されています．これは，オペランドのない命令ですが，他のレジスタのPUSH，POPと同様な動作をフラグ・レジスタに対してします．

フラグ制御命令の他にフラグ・レジスタを変更するために，LAHF（Load-AH-from-Flags），SAHF（Store-AH-into-Flags）命令が用いられます．これらはフラグ・レジスタの下位バイトを AH にロードする，またはストアする命令で，フラグ・レジスタの内容は図 4-6-5 のように AH 中に格納されます．

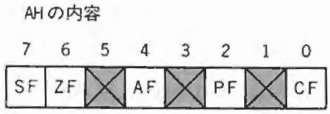


図4-6-5　LAHFの結果

LAHF，SAHF 命令を使ってフラグ・レジスタの内容を変えるには，リスト 4-6-2のようにします．これは，LAHF 命令によってロードしたフラグの任意のビットをOR 命令を使ってセットし，これを SAHF 命令でフラグ・レジスタへ戻します．この方法ですと，複数のフラグを一度に変更できるという利点があります．

リスト 4-6-2　フラグの変更

```
11BD:0100 B041        MOV    AL,41
11BD:0102 9F          LAHF ------------ フラグの内容をAHに転送
11BD:0103 08C4        OR     AH,AL ---- AHとAL(41H=01000001B)との和をAHに入れる
11BD:0105 9E          SAHF ------------ AHの内容をフラグに転送し，結果としてZFとCFがセットされる
11BD:0106 90          NOP
```

4-6-4　CPU制御命令

8086 のマシン語の中で CPU の制御に関する命令がいくつかあります．

HLT（Halt）命令は，CPU の動作を停止させてしまうための命令です．この命令を実行したあと，CPU を再び動作させるには，リセットをするか，外部割込みをかけなければなりません．

NOP（Non-Operation）命令は，その名の通り何も行わない命令です．この命令があると CPU は，IP を 1 進めただけで次の命令に実行を移します．NOP 命令は，デバッガなどによるプログラム開発において意外と役に立ちます．プログラムの開発中に，不確定な部分があるとき，その部分に NOP 命令をいくつか入れておきます．すると，たとえば数バイト命令がふえても，NOP 命令を消してその上に書くだけで，その後のすべてを書き直すような必要がなくなります．

その他の CPU 制御命令として，WAIT（Wait），ESC（Escape），LOCK命令があります．これらは，8086 CPU のバス（データ，アドレスの信号線）を共用しうる，他のバスマスタ（BUS Master）のための命令です．PC-9801 では数値演算プロセッサ 8087 がこれにあたります．

WAIT 命令は，8086 が動作を止め，他のバスマスタへシステムの制御を渡し，TEST 信号が渡されるまで待ち状態を保つ命令です．8087 の Busy 出力がこの Test 信号につながれており，8086 が 8087 に制御を渡すときはこの WAIT 命令を使用し，8087 の処理を終えたら，TEST 命令によ

って制御を戻します．

ESC 命令，LOCK 命令も 8086 のマルチ CPU システムのために用意された命令で，8087 を使用しない PC-9801 上では使用することはまずないと思われます．ハードウェアに興味のある方は参考文献を参照して下さい．

4-6-5 ストリング命令

これまで，ストリング命令として MOVS，CMPS，SCAS という命令について説明しました．ここで，ストリング命令についてまとめておきましょう．ストリング命令の共通点は SI，DI を専用のポインタとしてメモリをアドレッシングすることで，これはその名の通り文字列（ストリング）の処理のために用意された命令です．

ストリング命令にはこの他に LODS(Load-bytes-or-word-String)，STOS(Store-bytes-String-or-store-word-String) 命令の 2 つがあります．

LODS 命令は，SI で示されるメモリ中のバイト，またはワードデータを AL，または AX にロードし，DF の値によって SI の値を 1 または 2 ずつインクリメント，デクリメントします．

逆に，STOS 命令は，AL または AX のデータを DI で示されるエクストラ・セグメント内のメモリに格納する命令で，同じく DF の値により DI の値を 1 または 2 インクリメント，デクリメントします．

表 4-6-2 に，すべてのストリング命令についてまとめました．また，**リスト 4-6-3** に LODS，STOS 命令の例を示しましたので実行してみて下さい．

リスト 4-6-3 小文字／大文字変換ルーチン

```
11BD:0100 8CD8       MOV    AX,DS     ] DSとESを等しくする
11BD:0102 8EC0       MOV    ES,AX     ]
11BD:0104 FC         CLD  ─────────── ストリング命令のポインタを増加方向にする
11BD:0105 BE0002     MOV    SI,0200   ] データエリアの先頭アドレス
11BD:0108 BF0003     MOV    DI,0300   ]
11BD:010B B91000     MOV    CX,0010 ─ カウンタ，114Hとの間を16回繰り返す
11BD:010E AC         LODSB ────────── SIの示すアドレスの内容をALに転送
11BD:010F E80500     CALL   0117 ──── サブルーチンコール
11BD:0112 AA         STOSB ────────── DIの示すアドレスにALの内容を転送
11BD:0113 49         DEC    CX
11BD:0114 75F8       JNZ    010E
11BD:0116 90         NOP
11BD:0117 3C61       CMP    AL,61     ]
11BD:0119 7206       JB     0121      ]
11BD:011B 3C7A       CMP    AL,7A     ] ALの値が61Hより大きく7AHより小さければ
11BD:011D 7702       JA     0121      ] 20Hを引く
11BD:011F 2C20       SUB    AL,20     ]
11BD:0121 C3         RET ──────────── メインルーチン(112H)に戻る
```

表4-6-2　ストリング命令のまとめ

ストリング命令	ニーモニック	機　能	メモリの指定方法と命令の動作
MOVS	MOVSB	バイトストリングのメモリーメモリ間転送	ES：[DI]←DS：[SI] DF＝0：DI＝DI＋1，SI＝SI＋1 DF＝1：DI＝DI−1，SI＝SI−1
	MOVSW	ワードストリングのメモリーメモリ間転送	ES：[DI]，[DI＋1]←DS：[SI]，[SI＋1] DF＝0：DI＝DI＋2，SI＝SI＋2 DF＝1：DI＝DI−2，SI＝SI−2
CMPS	CMPSB	バイトストリングのメモリーメモリ比較	ES：[DI]−DS：[SI]→フラグ・レジスタ DF＝0：DI＝DI＋1，SI＝SI＋1 DF＝1：DI＝DI−1，SI＝SI−1
	CMPSW	ワードストリングのメモリーメモリ比較	ES：[DI]，[DI＋1]−DS：[SI]，[SI＋1]→フラグ・レジスタ DF＝0：DI＝DI＋2，SI＝SI＋2 DF＝1：DI＝DI−2，SI＝SI−2
SCAS	SCASB	ALとメモリ中のバイトストリングの比較	AL−ES：[DI]→フラグ・レジスタ DF＝0：DI＝DI＋1 DF＝1：DI＝DI−1
	SCASW	AXとメモリ中のワードストリングの比較	AX−ES：[DI]，[DI＋1]→フラグ・レジスタ DF＝0：DI＝DI＋2 DF＝1：DI＝DI−2
LODS	LODSB	メモリ中のバイトストリングのALへのロード	AL←DS：[SI] DF＝0：SI＝SI＋1 DF＝1：SI＝SI−1
	LODSW	メモリ中のワードストリングのAXへのロード	AX←DS：[SI]，[SI＋1] DF＝0：SI＝SI＋2 DF＝1：SI＝SI−2
STOS	STOSB	AL中のバイトストリングのメモリへの格納	AL→ES：[DI] DF＝0：DI＝DI＋1 DF＝1：DI＝DI−1
	STOSW	AX中のワードストリングのメモリへの格納	AX→ES：[DI]，[DI＋1] DF＝0：DI＝DI＋2 DF＝1：DI＝DI−2

5章
アセンブラによるマシン語の開発

第4章では，いくつかのマシン語プログラムを作成することを通して，8086の命令の特徴と使用法について解説しました．ここまでは，各命令の動作の確認が主な目的でしたので，プログラム作成のためのツールとして，DEBUGのワンライン・アセンブラを使用してきましたが，本来DEBUGは，本格的なアセンブラによって作成したプログラムのデバッグに使用するもので，プログラム開発よりも，データの部分的な修正やプログラムのテストに便利なように作られています．そのため，デバッガを使っていたのでは，プログラムの一部を変更するために，変更点以降のプログラムをすべて入力しなおさなくてはならないことがしばしばおこります．

そこでこの章では，より本格的なマシン語プログラムの開発のために，MS-DOSのMASM．EXEというマクロ・アセンブラについて，その基本的な特徴と使用の手順，プログラムの書き方を説明したうえで，第4章で紹介したサンプル・プログラムをMASM用に記述する実例を紹介します．

なお，ここで解説するMASMの起動には，最低256KバイトのRAMが必要です．

5-1 MASMの概要

5-1-1 MASMの特徴

ここまで読まれた方は理解されたでしょうが，マシン語で，何か有用なプログラムを作るには，数十から数百バイト，ときには千バイト以上の命令を記述しなければなりません．

たとえ，多少なりとも人に理解しやすい，アセンブリ言語を使用したとしても，大変な労力です．しかも，デバッガを使用していたのでは，入力したニーモニックが片端からマシン語に変換され，消えていってしまううえ，プログラムの意味を記すためのコメントも，分岐先を示すラベルという便利な記号も使用できません．

そこで，コメントやラベルを使ってニーモニック表現したプログラム（ソース・プログラム）をファイルにしておき，コメントやラベルを実際の値に変換しながらまとめてマシン語に変換する，アセンブルという手段が考え出されました．

このとき使用されるプログラムが，アセンブラ・プログラムで，これは，アセンブリ言語（ニ

ーモニック）で記述されたソース・プログラムをマシン語コード（オブジェクト：Objectという）に変換するものです．

端的に述べれば，アセンブリ言語で記述されたソース・プログラムをアセンブラ・プログラムによってアセンブルし，オブジェクト・プログラムを得るのです．これらの用語はよく混同されるので，あらためて記しておきます．

PC-9801のMS-DOSに標準で用意されているMASM．EXEは，このようなアセンブラの中でもマクロ・アセンブラと呼ばれるもので，8086の能力を十分に活かすために大型機並みの強力な機能を含んでいます．

たとえば，一般のアセンブラの機能は次のようなものです．

○プログラムの分岐先や，アドレスの値を示すのにラベルを使用することができる
○定数や変数を定義して使用することができる
○16進だけでなく，10進，2進，8進などさまざまな数値表現ができる
○プログラム中にコメントを入れることができる
○ソースファイルが残る

MASMには，これらの他に，

○マクロ機能によりソース・プログラムの短縮化ができる
○条件アセンブルができる
○分割アセンブルができる
○セグメント管理に関しての擬似命令が多く用意されている
○リロケータブル（再配置可能）なオブジェクト・コードの生成ができる
○オブジェクト・コードは，MS-LINK（リンカ）によってライブラリルーチンと組み合せて実行可能形式にすることができる

このうち，分割アセンブルとリロケータブルなオブジェクトコードの生成は，プログラムを機能ごとに分割して作成しておき，これを組み合せて目的のプログラムとするモジュール・プログラミングを可能とします．

ただ，これらの機能はMASMの使用法として高度な部類に属し，この特徴を十分に活かすには，ある程度熟練していることが必要です．本書では，これらの機能についてはさらに高度な使用法としてAPPENDIXでまとめて述べることにして，この章では，MASMを使って入門者がプログラミングを行うための，最低限必要な知識について説明したいと思います．

5-1-2 アセンブルの手順

アセンブルの大まかな手順は，第1章でも述べましたが，MASMには，それ以外にもいくつかの手続きが必要です．

ここで，MASMを使ってソース・プログラムから実行可能なマシン語プログラムを得るための手順の概略を述べておきます．

図5-1-1に，アセンブルの手順の流れを示しました．

この図の流れに沿って，各部分の働きを簡単に解説しましょう．

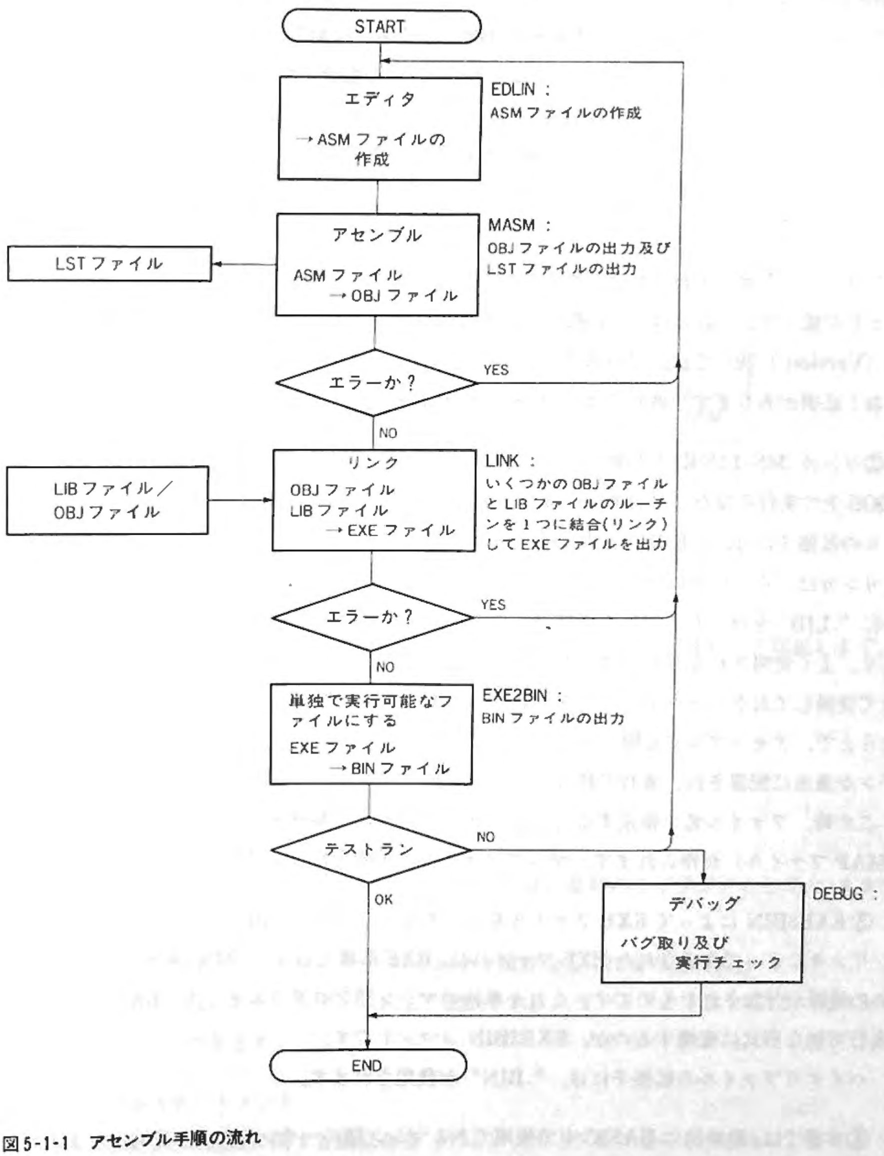


図5-1-1　アセンブル手順の流れ

テキスト・エディタ（後述）を使って，アセンブリ言語で記述したソース・プログラムをファイル名に".ASM"という拡張子を付けてディスクに格納しておきます．このファイルをソースファイルと呼びます．

①MASMを起動して，オブジェクトファイル(OBJファイル)を作成します．オブジェクト・ファイルネームには格納したソースファイルのファイル名の部分のみを指定します．オブジェクトファイルには，".OBJ"という拡張子が自動的に付加されます．

またMASMは，ソースの変換だけではなく，きちんと変換されたかどうかの確認のために，ソース・プログラムとマシン語を対応させて見ることのできるリスティングファイル（LSTファイル）を作成する機能を持っており，必要があればファイル名を入力します．リスティングファイルには，".LST"という拡張子が付加されます．

さらに，クロスリファレンスファイル（CRFファイル）という，ソース・プログラム中で使用したラベルや変数，定義された行や，参照されている行の一覧が記述されたファイルを出力することも可能です．これには，".CRF"という拡張子が付加されます．ただしこれを使うには，現在（Version 1.20）では，ソースファイルの先頭で，セグメント関係のきちんとした記述を行っておく必要があります（詳しくは，「MS-DOS v2.0マクロアセンブラマニュアル」参照）．

②リンカ(MS-LINK)を起動してMASMによって出力されたオブジェクト・ファイルを，MS-DOS上で実行可能なエグゼキュートファイル（EXEファイル）にします．エグゼキュートファイルの拡張子には，".EXE"が付加されます．

リンカは，それ以外に複数のサブルーチン・プログラム（ライブラリファイルと呼ばれ，拡張子に".LIB"を伴ったもの）をオブジェクトファイルに結合させる機能を持っています．これにより，よく使用されるプログラムをサブルーチンとしてモジュール化し，ライブラリファイルとして登録しておき，ソース・プログラム中で，あたかも内部のサブルーチンを呼ぶように記述したうえで，アセンブルする際にリンカ内でこれらのモジュールを指定すれば，自動的にそのルーチンが適当に配置され，実行可能なひとつのプログラムにしてくれるのです．

この時，ファイル名を指定することで，外部ルーチンの参照状態が記述されたマップファイル（MAPファイル）が作られます．マップファイルの拡張子には".MAP"が付加されます．

③EXE2BINによってEXEファイルをバイナリ・ファイル（BINファイル）にします．

リンカによって作成されたEXEファイルは，EXE形成といって，MS-DOS上で実行されるための情報が付加されたものです．これを単独のマシン語プログラムとして（BASIC環境内でも）実行可能な形式に変換するのが，EXE2BINコマンドです．

バイナリファイルの拡張子には，".BIN"が使用されます．

④本書では，最終的にBASIC中で使用できるマシン語ルーチンの作成を目的としますから，作業はここで終わりです．しかし，MS-DOS上でコマンドとして利用するようなマシン語ルーチン

の場合は，さらにBINファイルをリネームしてファイルの拡張子の".BIN"を".COM"に変更し，コマンドファイル（COMファイル）にする必要があります．なぜなら，拡張子に".EXE"か，".COM"を伴ったファイルのみが，MS-DOSのコマンドとして許されたものだからです．

さて，ここで得られたBINファイルは，MS-DOSから切り離して使用することができます(ただし，バグがないとしての話ですが)．BINファイルをBASICファイルにする方法と，BASICプログラム中でサブルーチンとして利用する方法は次章で解説することにして，ここでアセンブル中に出てきたMASM特有のファイル名についてまとめておきましょう（表5-1-1）．

表5-1-1　アセンブルで使用されるファイルの種類

ファイルの種類	意　　味	拡張子
ASMファイル	ソースファイル	.ASM
OBJファイル	MASMによって出力されるファイル	.OBJ
EXEファイル	LINKによって出力されるファイル (MS-DOS上で実行可能)	.EXE
BINファイル (COMファイル)	EXE2BINによって出力されるファイル (独立して実行可能)	.BIN (.COM)

5-2　MASMの用法

5-2-1　メモリ擬似命令

この節では，MASMを使用する場合の，アセンブラ・プログラムの記述形式を解説します．

ソース・プログラムの記述に使用できる文字は次の通りです．

○英字　　A～Z（大文字），a～z（小文字）

○数字　　0～9

○特殊記号　?，@，_，$など

また，ラベルやネームなどの名前フィールド（後述）には，最初のキャラクタとして0~9までの数字は使えません．

逆に，数値を書く際は，すべて頭に0~9のどれかを置かなくてはなりません．

ソース・プログラムは，ステートメントにより記述され，ステートメントはさらに，次の3つに分けて考えることができます．

① 8086の命令ステートメント

② 変数，定数などの定義に関する擬似命令（ディレクティブ）ステートメント

③ アセンブルする際の制御に関する擬似命令（ディレクティブ）ステートメント

このうち，MASMの記述形式を特徴づけているのが，③のアセンブルに関する制御のための擬似命令です．

擬似命令とは，ニーモニック命令のようにマシン語化される命令ではなく，アセンブルの際のメモリ指定や，アセンブルの条件等の制御に使用される命令のことで，制御のための擬似命令としてはメモリ管理を行うメモリ擬似命令，マクロ機能のためのマクロ擬似命令，条件擬似命令，リスティング擬似命令等があります．

このうち，マクロ命令以降の命令は，かなり大がかりなプログラムの作成に威力を発揮する命令で，入門者としては，メモリ擬似命令までを知っておけば十分です．ここでは，この擬似命令について説明します．

メモリ擬似命令には，**表5-2-1** に示すように使用するセグメントをアセンブラに対して知らせるものやプログラムの終りを示すものなどがあります．

表5-2-1　メモリ擬似命令と定義擬似命令

	名称	用法
メモリ擬似命令	ASSUME	セグメント・レジスタがどのセグメントを指すかをアセンブラに知らせる
	ASSUME NOTHING	暗黙指定のセグメント・レジスタをなくし，すべて明示により動作する
	COMMENT	区切り記号ではさまれた文字列をコメントと見なすようにアセンブラに知らせる
	SEGMENT～ENDS	この擬似命令にはさまれたソースをこのセグメント内のコードとする
	ORG	ロケーション・カウンタの値をセットする
	END	ソースステートメントの終わり
定義擬似命令	EQU	名前に数値を割り当てる
	=	再定義できるEQU擬似命令
	DB, DW, DD, DQ, DT	それぞれの型の変数を定義する
	LABEL	コード・セグメントに仮定されているアドレスに名前を付ける

MASMにおけるプログラムはひとつのセグメントの中に入っており，これをSEGMENT擬似命令で指定します．これは，ENDS擬似命令と組になって使用し，SEGMENT，ENDSの前にはそのセグメント名を書きます．

```
Main  SEGMENT
      ASSUME  CS：Main，DS：Main
        ⋮
      プログラム本体（命令ステートメント）
        ⋮
Main  ENDS
      END
```

この例では，セグメント名はMainであり，実際のプログラムはこの中に書かれます．記述は大文字でも小文字でもかまいません．

そしてプログラムの最後には，必ずEND擬似命令を指定します．これを忘れるとアセンブル時にエラーとなります．

SEGMENT擬似命令の次には，ASSUME擬似命令を使ってセグメント・レジスタが，どのセグメントを指すかをアセンブラに知らせます．ここでは，CS, DSともにセグメント名Mainを指定しました．アセンブラは，実行時のセグメント・レジスタの値はわかりません．つまり，アセンブル時にプログラムのオペランドに書かれたオフセットがどのセグメント・レジスタを使用しているのかの情報がないのです．

そこで，ユーザーはあらかじめ使用するセグメント・レジスタとその値をASSUMEを使ってアセンブラに予告しておきます．アセンブラはこれを見てセグメント・オーバーライド・プレフィックスを決めマシン語コードを作るので，マシン語プログラムの中では，予告した通りの値を各セグメント・レジスタにセットしなくてはなりません．プログラム途中でセグメントを切り換える場合は，このASSUMEをソース・プログラムの途中に挿入することで行います．

メモリ命令には，この他にORGがあります．これは，プログラム中で，マシン語コードや変数が実際に配置されるアドレスを指定するときに，その直前に置かれます．そして，このORGのオペランドには，ASSUMEで決められたセグメントからのオフセット値を指定します．

アセンブル制御に関する命令は，この他にも数多く用意されていますが，最低限以上のことを知っておけば，通常のプログラムを組むうえでは十分です．

5-2-2　命令ステートメントと定義擬似命令

ここでは，プログラムの中身である命令ステートメント部分について解説しましょう．命令ステートメントは，さらにいくつかの部分（フィールド）に分かれており，その間は，1個以上のスペース（空白）またはタブ（TAB）で区切られます．

ステートメントは次のように5つのフィールドに分けられます．

```
ラベル：　リピート命令　ニーモニック　　オペランド　　；コメント
例）
BEGIN：　　REP　　　　　MOVS　　　　DEST，SORC　；メモリーメモリ間データ転送
```

このうちニーモニックは，必ず書かなくてはいけませんが，その他のフィールドは必要に応じて指定します．

コメントは，先頭にセミコロン（；）を付加することでどこにでも書け，BASICのREM文と同じように，命令の実行には関係のない，ソース・プログラム中での覚書きなどに使います．

```
例）
; This is a machine language program.
```

アセンブラは，これらのコメントはすべて取り除いてOBJファイルを作ります．

命令ステートメントの，ニーモニックとオペランドの記述形式のうち，以下のものはデバッガで使用したのと同じです．

○レジスタの表現（表3-1-1参照）

○角カッコによるアドレッシングモードの指定

○セグメント・オーバーライド・プレフィックス指定

デバッガでの表現形式との違う点は以下のものがあります．

○オペランドは式で表現でき、式中には算術演算子，論理演算子，関係演算子が使用できる（表5-2-2）

○数値表現は、16進数以外に2進，8進，10進数が表せる．このとき16進数では数値の先頭に必ず0～9のどれかを置くようにする（表5-2-3）

表5-2-3　アセンブラ中の数値表現

基　数	表記法，注意	例
2進数	××××××××B	01001100B
8進数	×××O ×××Q	421O 326Q
10進数	××××× ×××××D	32512 65535D
16進数	××××H （初めの桁は0～9）	0FFE0H 2F30H
10進実数	××．××E±××	37.38E－22
16進実数	×……×R	2F5A3BC1R
ASCII文字	'××' "××"　（3文字以上定義できるのはDBのみ）	DB 'ASCII' DW 'MI'

○分岐命令の分岐先アドレスは，ラベル名で指定する．JMP命令では特にShort（－128～127），Near（－32768～32767），Far（セグメント間）などの属性を指定する．指定がない場合はNearになる

○ストリング命令の繰返しを行うREP命令は，繰返しプレフィックスとしてストリング命令の前に書き，デバッガでの入力のように別々の行には記述できない（図5-2-1）

○ストリング命令のオペランドとして，データのタイプのチェック（ワードかバイトか）のために、変数名を指定することができる

表5-2-2　アセンブラで使用できる演算子

	演算子	意　味	例
算術演算子	+	加　算	MOV AX, 20H＋41H (MOV AX, 61Hと同じ)
	−	減　算	MOV BX, 40−5 (MOV BX, 35と同じ)
	*	乗　算	MOV BX, 80*12 (MOV BX, 960と同じ)
	／	除　算	ADD CX, 0FFH/0FH (ADD CX, 11Hと同じ)
	MOD	剰　余	MOV AL, 31 MOD 2 (MOV AL, 1と同じ)
	SHR	右シフト	MOV AL, 01001000B SHR 2 (MOV AL, 00010010Bと同じ)
	SHL	左シフト	MOV BL, 00001101B SHL 4 (MOV BL, 11010000Bと同じ)
論理演算子	AND	論理積	ADD AX, 0F1H AND 1FH (ADD AX, 11Hと同じ)
	OR	論理和	CMP BL, 0F0H OR 44H (CMP BL, 0F4Hと同じ)
	NOT	否　定	MOV AL, NOT 80H (MOV AL, 7FHと同じ)
	XOR	排他的論理和	MOV AL, 41H XOR 52H (MOV AL, 13Hと同じ)
関係演算子	EQ	等しい	これらは，主に条件アセンブルに必要な条件擬似命令で使われます ２つのオペランドの関係が真のときFFFFHが返され，偽のとき０が返されます 例） IF BYTE_DATA EQ 0A0H
	NE	等しくない	
	LT	より小さい	
	LE	より小さいか等しい	
	GT	より大きい	
	GE	より大きいか等しい	
アトリビュート（属性）演算子	SEG	変数，ラベルのセグメント値を得る	MOV DX, SEG LABEL1
	TYPE	変数，ラベルのタイプを数として得る	MOV AL, TYPE VAR1 (1:BYTE, 2:WORD, 4:DWORD, 6:QWORD, 10:TWORD, STRUCのバイト数)
	OFFSET	変数，ラベルのセグメントベースからのオフセットを得る	MOV AL, OFFSET VAR2
	SIZE	変数が占める全バイト数	ADD SI, SIZE VAR3
	LENGTH	変数の長さを取り出す LENGTH×TYPE＝SIZE	MOV AX, LENGTH VAR4
	PTR	一時的にオペランドのタイプを変える	MOV AL, BYTE PTR WORD_VAR
	THIS	名前に現在のセグメントとオフセットをつけ，タイプをつける（変数，ラベルの生成）	NEW_VAR EQU THIS BYTE
	HIGH	ワード変数のオフセットの上位バイトを返す	MOV AH, HIGH WORD_VAR
	LOW	ワード変数のオフセットの下位バイトを返す	MOV AL, LOW WORD_VAR
	CS: DS: ES: SS:	セグメントオーバーライド，暗黙のセグメント指定の代わりに，指定したセグメントを使う	MOV AX, CS: [BX]
	SHORT	オペランドを１バイトオフセットにする	JMP SHORT LBL1

デバッガ表記	マクロアセンブラ表記
	SOURCE DB 'ABCDEFGHIJKLMNOP'
	DEST DB 10H DUP(?)
MOV SI,100	MOV SI,OFFSET SOURCE
MOV DI,200	MOV DI,OFFSET DEST
MOV CX,10	MOV CX,10H
REPZ	REP MOVS DEST,SOURCE
MOVSB	(またはREP MOVSB)

図5-2-1 ストリング命令の表記の違い

一方，ステートメント中で使用する変数は，定義擬似命令（デファイン・ディレクティブ）ステートメントによって定義することができます．

定義擬似命令ステートメントは，次のように4つのフィールドで構成されます．

ネーム　定義擬似命令　式フィールド　；コメント

例）

```
VAR1    DW    20H    ;2バイトの変数VAR1を初期値20Hとして定義
```

これも同様に，定義擬似命令以外は，必要に応じて指定します．

定義擬似命令によって指定した領域は，アセンブルによってメモリ内の破壊されない部分にとられ，最初は式フィールドの内容が書き込まれます．そして，プログラム中に記されたネーム名（VAR1等）には，そのアドレスが代わりに書き込まれます．

初期値が任意でよい場合は，式フィールドに"?"を書いておきます．

定義擬似命令によって指定できる変数の長さには，DB（バイト），DW（ワード），DD（ダブルワード＝4バイト），DQ（8バイト），DT（10バイト）があります．

同様の定義命令としては，EQU命令，"＝"があります．これらは定数の定義に使用され，

```
FIXD1   EQU  10
FIXD2   =    20H
```

のように記述されますが，これらの命令によって定められた定数値は，アセンブルの際に，どちらもリスト中で定数名を参照している部分に置き換わるだけです．

ただし，"＝"を使用した変数の場合は，プログラム中で再定義が可能です．

MASMの記述方式について，最小限の部分にしぼって解説しました．簡単なアセンブラ・プログラムを作成するうえでは，この程度でも可能ですが，さらに高度な利用を考えている方は，「MS-DOS マクロアセンブラマニュアル」を読まれるとよいでしょう．

5-3　MASMによるプログラミング

5-3-1　EDLINの使用法

アセンブリ言語のソース・プログラムを作成するいちばん手軽な方法は，MS-DOSに標準で用意されているラインエディタ(EDLIN．EXE)を使うことです．EDLINは，BASICにあるようなスクリーンエディタとは異なり，文章またはプログラムの編集を1行単位で行うものです．

表5-3-1に，EDLINのコマンド一覧を示します．ここでは，これらのコマンドのうち，アセンブラ・プログラムファイルを作るのに十分なコマンドの使い方について述べます．EDLINについての詳細は「MS-DOSユーザーズマニュアル」を参照して下さい．

表5-3-1　EDLINのコマンド一覧

コマンド	目　的
〈行番号〉	編集する行の指定
A	行の付加
C	行のコピー
D	行の削除
E	編集の終了
I	テキストの挿入
L	テキストのリスト
M	行の移動
P	テキストのページ化
Q	編集の取消し
R	テキストの置換え
S	テキストの検索
T	テキストの読込み
W	行の書込み

EDLINの起動は，次のように編集を行うファイルの名前を指定して行います．

```
A>EDLIN  Program．ASM⏎
```

ファイルが新規作成のものであるときには，

```
新しいファイルです
*■
```

と表されます．また，すでに存在するファイルへの書き込みの場合は，

```
ファイルを終りまで読み込みました
*■
```

と表示され，コマンド入力待ちを知らせるプロンプト（*）とカーソルが表示されます．

EDLINを終了してMS-DOSの入力待ち状態へ戻るには，2つのやり方があります．入力した内容をディスクに格納してから終えるときは，E (Exit) と入力します．ファイルへの格納をしないときにはQ (Quit) コマンドを入力します．"Q" と入力すると，

編集を中止しますか<Y/N>？

と聞いてくるので，"Y"を入力すればMS-DOSにもどります．また"N"を入力すると，EDLINのコマンドレベルにもどります．

"E" を入力した際に書き込まれるのは，EDLINの起動時に指定したファイルです．

新しいプログラムの入力を行うには，I (Insert) コマンドを行います．プロンプト表示状態で"I"と入力すると次のように行番号が表示され，プログラムの入力が可能になります．この状態をインサートモードといいます．

```
*I⏎
          1：*
```

行番号の次のアスタリスクは現在エディタが着目している行を示しています．

ここでプログラムの入力を行います．入力ミスをしたときには，BS (バックスペース)，DEL (デリート)，または左カーソルキーのいずれかを使って文字を消し，入力し直します．1行の入力の終了後はリターンキーを押せば，次の行の番号が表示されます．以下同じようにして入力を行います．

プログラム全体の入力が終了したら[CTRL]＋[C]を入力することでインサートモードから抜けます．

入力したプログラムが正しく入っているかどうかを確めるには，L(List)コマンドを使います．Lと入力すれば現在着目している行の前後23行分が表示され，着目行にはアスタリスクがつきます．1行目からの表示には"1L"のように行番号を指定します．Lコマンドでは着目行を移動しませんので，Lを何度入力しても表示される範囲は同じです．

一方，P (Print) コマンドを用いれば，着目行は表示されたリストの最終行へ移ります．Pコマンドを行うことでリストは23行ずつ順に見ることができます．

EDLINのコマンドレベルでリターンのみを入力すると，現在着目している次の行が表示されて，その行の変更モードになります．変更の必要のない時はリターンのみを入力すれば，EDLINコマンドレベルに抜けます．

着目行の移動には行番号のみを入力します．すると，リターンのみを入力したのと同じように，その行の変更モードとなります．

各コマンドに行番号を指定するには，そのコマンドの前に行番号を入力します．いくつかの例を示します．

5I	5行目の前から行を挿入する
30L	30行目の前後23行のリスト
10，30L	10行目から30行目までのリスト
10P	10行目から23行分つまり32行目まで表示され，着目行は32行に移る

行の削除にはD（Delete）コマンドを使います．例えば5行目を削除するには，

5D

また，5行目から20行目までを削除するには，

5，20D

と入力します．Dコマンドの後は行番号はつめられます．

たいていのプログラム・ファイルは，ここで説明したコマンドにより作成できます．

なお，編集に便利な機能としてMS-DOSのテンプレート機能を使うことができます．これを表5-3-2に示しました．

表5-3-2　MS-DOSのテンプレート機能

コマンド	ファンクションキー	コード	機　能
1文字をコピー (Copy One character)	f･1	C1	新しい行にテンプレートから1文字コピーする
指定文字までコピー (Copy up to character)	f･2	CU	テンプレートから新しい行へ指定された文字の直前まで，すべての文字をコピーする
テンプレートをコピー (Copy All characters)	f･3	CA	新しい行に，テンプレート中に残っているすべての文字をコピーする
1文字をスキップ (Skip One character)	f･4	S1	テンプレート内の1文字をスキップする（コピーしない）
指定文字までスキップ (Skip up to character)	f･5	SU	指定された文字まで，テンプレート内の文字をスキップする（コピーしない）
入力の取消し (Void)	f･6	VOID	現在の入力を取り消す．テンプレートの内容は変化しない
テンプレートへの書込み (New Line)	f･7	NWL	現在行の内容で新しいテンプレートを作る
挿入モード (Insert mode)	f･8	INS	挿入モードに入るまたは挿入モードから出る
置換えモード (Replace mode)	f･9	REP	挿入モードから出る
CTRL＋Zの挿入	f･10	^Z	CTRL＋Zをテンプレートに挿入する

5-3-2　アセンブラによるプログラムの表記

ここで，MASMによるプログラム作成の実例として，第4章で紹介したバブルソート・プログラムをMASMの文法で表記したうえでEDLINを使って入力してみましょう．

リスト5-3-1は，バブルソート・プログラムのソースリストです．

リスト 5-3-1 MASM表記によるバブルソートプログラム

```
 1: ; bubble sort program
 2: ;
 3: data_adr EQU     200h
 4: nb_data  EQU     128
 5: Main     SEGMENT
 6:          ASSUME  CS:Main,DS:Main
 7:
 8:          ORG     100h
 9:
10: start:
11:          MOV     BX,data_adr
12:          MOV     DI,nb_data
13: ;
14: out_loop:
15:          DEC     DI
16:          CMP     DI,1
17:          JB      e_prog
18:          MOV     CX,DI
19:          DEC     CX
20:          MOV     SI,0
21: in_loop:
22:          CMP     SI,CX
23:          JA      out_loop
24:          MOV     AL,[SI][BX]
25:          MOV     AH,[SI][BX]+1
26:          CMP     AL,AH
27:          JBE     e_swap
28:          XCHG    AL,AH
29:          INC     count
30:          MOV     [SI][BX],Al
31:          MOV     [SI][BX]+1,AH
32: e_swap:
33:          INC     SI
34:          JMP     in_loop
35:
36: e_prog:
37:          RET
38:
39: count    DW      0
40:
41: Main     ENDS
42:
43:          END     start
```

○リストの解説

リストの1, 2行目はコメントのみの文です. 3行目はデータ領域のアドレス・オフセットを16進で定数として定義し, 4行目はデータの数を10進で定義しています. これらはEQU擬似命令を使用しています.

5行目からはプログラム・セグメントで, Mainというセグメント名をつけたSEGMENT擬似命令を書きます. これは41行目のMain ENDSと組になっています. SEGMENT擬似命令の次は, ASSUME擬似命令で, CS, DSともにMainにしています.

これは，MS-DOSのBINファイルが，実行の際すべてのセグメント・レジスタの値をCSと等しくしてから起動するのに合せたためで，このプログラムは，最終的にBINファイルとするため，SSも変更していません．

8行目は，プログラムのロード・アドレスをこのセグメントのベースから100H番地目からにするためのORG擬似命令です．

10行目はプログラムの始まりを示すラベルで，ここがプログラムのスタート・アドレスになります．

11,12行は，前にEQU擬似命令で定義した定数をBXおよびDIへセットしていますが，これはSORC部に定数名を書くことで行うことができます．

17行目は，ラベルを参照した条件分岐の例です．前のCMP命令の結果によりe_progというラベルへ飛びます．

24,25行および30,31行は，インダイレクトモードの表記方法です．

29行目は，変数を直接インクリメントする表記で，INC命令で，39行で定義したワード変数COUNTをインクリメントし，ソートのXCHG命令の実行回数を記録します．

34行目は無条件分岐命令です．

39行は，変数COUNTの定義をしており，DWにより値は0に初期化します．このように変数は同じセグメント中のプログラムの後ろにとっています．

41行はセグメントの終了であり，43行でアセンブラにプログラムの終了を指定するEND擬似命令とプログラムのスタート・アドレスを示すラベルを書き，ソースプログラムは終りです．

○プログラムの入力

プログラムの入力には，EDLINをファイル名を伴って起動します．ここでは"BUBSRT．ASM"としました．

```
A>EDLIN BUBSRT . ASM[↵]
```

EDLINのプロンプトが出たら，Iコマンドを入力します．

```
*I[↵]
        1:*■
```

ここで，一度TABキーを押してから，リスト5-3-1を入力します．

```
        1:*        ; bubble sort program[↵]
        2:*        ;[↵]
        3:*
```

入力は，各行ごとにTABを入れた方が後で美しく見えます．各行の行末でリターンキーを入力するごとに，次の行の入力をうながします．

入力を誤った時でも，リターンの挿入前ならば，バックスペースキーで修正できます．

```
41 : *      main      ends ⏎
42 : *⏎
43 : *                end       start ⏎
```

入力が終了したら，L または P コマンドで確認したうえ，E コマンドで EDLIN からぬけます(図 5-3-1)．

```
A>DIR BUBSRT.*

 ドライブ A: のディスクにはボリュームラベルがありません
 ディレクトリは A:¥

BUBSRT   ASM      489  85-01-31   4:37
        1 個のファイルがあります
   442368 バイトが使用可能です
```

図5-3-1 ディレクトリの確認

5-3-3 アセンブルの実際

前項で入力したバブルソート・プログラムのソースファイル BUBSRT．ASM をアセンブルして，実行可能なバイナリファイルを作る様子を以下に説明していきます．

まず，MASM を起動します．これには，DOS レベルから MASM と入力します．

すると図 5-3-2 に示すように MASM のオープニングメッセージが表示され，次の 1 行目にソースファイルの入力が促されます．ここでソースファイル名を入力します．

```
A>MASM
The Microsoft MACRO Assembler
Version 1.20, Copyright (C) Microsoft Inc. 1981,82,83

Source filename [.ASM]: BUBSRT
```

図5-3-2 MASM の表示 1

次に，オブジェクトファイル名の確認がなされます．デフォルトのファイル名は，ソースファイル名と .OBJ 拡張子です．

変更がなければリターンのみを押します．

次には，リスティングファイルの出力を行うかどうか聞いてきますので，必要があればファイル名を入力します．ここでは，ファイル名 BUBSRT を入力します．この結果リスティングファイル BUBSRT．LST が作成されます．

次には，クロスリファレンスファイルの出力の有無を聞いてきます．ここでは必要ないので単に，リターンキーを押します．

以上の入力が終わると，図のようにアセンブルの結果として，プログラムに対するエラーの数が出力されます（図5-3-3）.

```
A>MASM
The Microsoft MACRO Assembler
Version 1.20, Copyright (C) Microsoft Inc. 1981,82,83

Source filename [.ASM]: BUBSRT
Object filename [BUBSRT.OBJ]:
Source listing  [NUL.LST]: BUBSRT
Cross reference [NUL.CRF]:

Warning Severe
Errors  Errors
0       0
```

図5-3-3　MASMの表示 2

ここで Warning Error とは警告であり，Severe Error とは致命的なエラーをいいます．

正常にアセンブルが終了した場合は，図のようにエラーは0です．ここで出力されたエラーの生じた場所は，先に出力を指定したリスティングファイルに書かれています．この出力結果を図5-3-4に示します．

```
The Microsoft MACRO Assembler                 02-18-85      PAGE    1-1


                                  ; bubble sort program
                                  ;
= 0200                            data_adr EQU    200h
= 0080                            nb_data  EQU    128
0000                              Main     SEGMENT
                                           ASSUME  CS:Main,DS:Main

0100                                       ORG     100h

0100                              start:
0100  BB 0200                              MOV     BX,data_adr
0103  BF 0080                              MOV     DI,nb_data
                                  ;
0106                              out_loop:
0106  4F                                   DEC     DI
0107  83 FF 01                             CMP     DI,1
010A  72 21                                JB      e_prog
010C  8B CF                                MOV     CX,DI
010E  49                                   DEC     CX
010F  BE 0000                              MOV     SI,0
0112                              in_loop:
0112  3B F1                                CMP     SI,CX
0114  77 F0                                JA      out_loop
0116  8A 00                                MOV     AL,[SI][BX]
0118  8A 60 01                             MOV     AH,[SI][BX]+1
```

```
011B  3A C4                         CMP     AL,AH
011D  76 0B                         JBE     e_swap
011F  86 C4                         XCHG    AL,AH
0121  FF 06 012E R                  INC     count
0125  88 00                         MOV     [SI][BX],Al
0127  88 60 01                      MOV     [SI][BX]+1,AH
012A                        e_swap:
012A  46                            INC     SI
012B  EB E5                         JMP     in_loop

012D                        e_prog:
012D  C3                            RET

012E  0000                  count   DW      0

0130                        Main    ENDS

                                    END     start


 The Microsoft MACRO Assembler            02-18-85       PAGE     Symbols
                                     -1

Segments and groups:

                N a m e                   Size    align   combine class

MAIN . . . . . . . . . . . . . .          0130    PARA    NONE

Symbols:

                N a m e                   Type    Value   Attr

COUNT. . . . . . . . . . . . . .          L WORD  012E    MAIN
DATA_ADR . . . . . . . . . . . .          Number  0200
E_PROG . . . . . . . . . . . . .          L NEAR  012D    MAIN
E_SWAP . . . . . . . . . . . . .          L NEAR  012A    MAIN
IN_LOOP. . . . . . . . . . . . .          L NEAR  0112    MAIN
NB_DATA. . . . . . . . . . . . .          Number  0080
OUT_LOOP . . . . . . . . . . . .          L NEAR  0106    MAIN
START. . . . . . . . . . . . . .          L NEAR  0100    MAIN

Warning Severe
Errors  Errors
0       0
```

図5-3-4　リスティングファイル

次に，MASM の結果出力された BUBSRT . OBJ をリンカにかけます．この様子を図 5-3-5 に示します．

```
A>LINK

   Microsoft Object Linker V2.2(830822)
(C) Copyright 1982, 1983 by Microsoft Inc.

Object Modules [.OBJ]: BUBSRT
```

図5-3-5　LINKの表示 1

オープニングメッセージの後に，入力オブジェクトファイル名を入力すると，次の行に，出力されるエグゼキュート・ファイル名が表示され，確認がなされます．ここでリターンキーを入力すると，検索するライブラリファイル名の入力が促されます．

ライブラリファイルは必要ないため，リターンを入力します．すると図5-3-6のように，スタック・セグメントがないという警告がなされますが，BINファイルにはスタック・セグメントが必要でないため，無視してかまいません．

```
A>LINK

   Microsoft Object Linker V2.2(830822)
(C) Copyright 1982, 1983 by Microsoft Inc.

Object Modules [.OBJ]: BUBSRT
Run File [BUBSRT.EXE]:
List File [NUL.MAP]:
Libraries [.LIB]:
Warning: No STACK segment

There was 1 error detected.
```

図5-3-6 LINKの表示 2

以上の結果，リンカによりBUBSRT．EXEファイルができました．

最後にEXE2BINコマンドにより，EXEファイルをバイナリファイルBUBSRT．BINに変換して一連の操作は終了です．この後デバッガによりデバッグを行い，論理的なバグを取りプログラムを完成させます（図5-3-7）．

```
A>EXE2BIN BUBSRT

A>DIR BUBSRT.*

 ドライブ A: のディスクにはボリュームラベルがありません
 ディレクトリは A:¥

BUBSRT   ASM      495  85-02-18   5:03
BUBSRT   OBJ      102  85-02-18   5:36
BUBSRT   LST     1910  85-02-18   5:36
BUBSRT   EXE      816  85-02-18   5:36
BUBSRT   BIN       48  85-02-18   5:38
        5 個のファイルがあります
   315392 バイトが使用可能です

A>
```

図5-3-7 ファイル名の確認

6章
BASICとマシン語プログラムのリンク

これまでの各章ではPC-9801のCPU(8086)とそのマシン語について説明してきました．これでマシン語プログラムに関する基礎知識は理解できたと思います．しかし，実用的なソフトウェアをマシン語プログラムのみで作るには大変な作業がいります．そこでこの章では，実用的なマシン語プログラムの利用法として，BASICとマシン語プログラムを組み合わせて，1つのプログラムにする方法を説明します．またBASIC-ROM内のマシン語サブルーチンの利用法についても説明します．

6-1　マシン語の有効利用

6-1-1　BASICとリンクすることの有利さ

これまでみてきたように，マシン語プログラムを作成するには，たとえアセンブラを使ったとしても大変な時間と労力を必要とします．とくに実用的なソフト，たとえばワードプロセッサや集計表，またはゲームなどの大きなプログラムをマシン語のみで作ろうとしたら，そのための開発時間は，これまで例にあげたサンプル・プログラムとは比べものにならないほど膨大になるでしょう．なぜなら，実用的なソフトでは本質的な部分だけではなく，たとえばキーボードから入力した文字や数字を取り込むルーチンや，CRTにメモリの内容を表示するルーチンなどの基本的なプログラムをも，作成しなければならないからです．

そこで考えられるのはBASICプログラムとマシン語プログラムとのリンク(Link：組み合せ，結合）です．つまりプログラムのマンマシン・インターフェイス（Man-Machine Interface）を含むI/O(入出力)制御に関する部分はBASICで記述し，マシン語プログラムに適した部分をマシン語で記述して，これをBASICプログラムからサブルーチンコールによって使用するわけです．

この方法はBASICだけに限らず一般の高級言語(Pascal，C，etc……)についてもいえることで，アマチュアのプログラミングには，有効な手段であるといえます．ここではBASICを例にとって説明していきます（図6-1-1)．

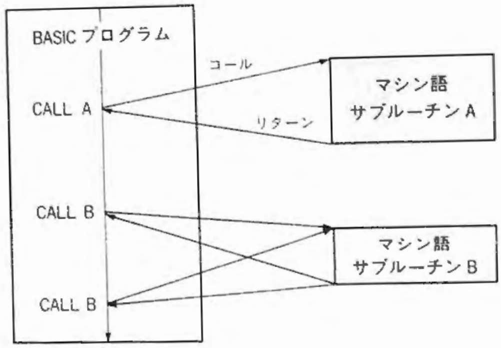


図6-1-1　マシン語サブルーチンコールの概念図

BASIC とマシン語プログラムをリンクすることの利点のひとつは，一般の I/O 制御を BASIC で記述できることで，これにより柔軟性を持ったプログラムが書け，信頼性も上がります．またデバッグも早く行え，プログラムの開発時間を短縮でき，さらに，マシン語ルーチンも機能ごとに単体でテストできるので，デバッグの点でも有効です．

欠点としては，すべてをマシン語で書いたプログラムに比べて実行速度が遅くなることが考えられますが，しかしこれはどんな場合にもそうなるとは限りません．例えばキーボードからの入力や，CRT への文字の表示などの，人間と直接，接する部分（マンマシン・インターフェイス）は，人の反応が遅いので BASIC による処理でも問題はないでしょう．

また，初期設定などプログラム実行中に数回しか行わない作業などは，BASIC で記述する方が簡単ですし，プログラムの見通しがよくなります．

このように，BASIC とマシン語のリンクでは，プログラムのどの部分をマシン語にするかがプログラムを能率良く作成するポイントになるでしょう．

6-1-2　何をマシン語化するか

では，プログラムのどこをマシン語化したらよいのでしょうか．それにはマシン語の利点を思い出してみましょう．

① プログラムサイズを小さくでき，実行速度の速いプログラムがつくれる
② BASIC ではできない，ハードウェアに直結した機能を実現できる
③ CPU の機能を生かしたプログラミングができる

マシン語の有効利用は，これらの利点を活かすような部分をマシン語に置きかえればよいのです．具体的には，①の利点を活かすには BASIC プログラムのなかで何度も繰り返される計算や処理を，②の利点を活かすには，A/D コンバータ（アナログ／ディジタル変換器）などの周辺 I/O デバイスの制御プログラムを，③の利点を活かすには，8086 に特有の命令を使用した作業（たとえ

ば，ストリング命令を使ったデータのソーティング等）を，それぞれマシン語ルーチンにするわけです．

これとは反対にプログラム中でマシン語化すべきでない部分，すなわちBASICの利点を活かすべき部分を挙げてみましょう．

BASICの最大の利点は，プログラムの変更が対話形式（インタラクティブ）に行えるということです．これはとくにハードウェア，例えばスクリーン制御で表示画面の設定などのパラメータを変更するときなどに大変便利です．

これをマシン語で行うとしたら，パラメータのデータ構造および画面設定のハードウェアについての知識が必要になる上，試しながら設定値を決めるなどの対話的な変更が難しくなります．

このように，我々がマシン語を用いてプログラムを作るときには，すべてをマシン語化することよりも，まず，何をしたいかをよく考え，それをいくつかの機能に分け，それぞれの機能の性質によりBASIC（または他の高級言語）で記述するか，マシン語にするかを決めるのが効率の良い，賢いやり方です．そして，そのためにもBASICの見直しが必要となります．

6-2　BASICとメモリマップ

6-2-1　N_{88}-BASIC(86)のメモリマップ

BASICプログラムからマシン語サブルーチンを呼ぶには，そのマシン語プログラムをメモリ上にロードしなくてはなりません．これまで学んできたように，マシン語プログラムはCPUのメモリ空間に置かれて実行されますが，これをBASICと共存させるには勝手なアドレスに配置するわけにはいきません．BASIC自身もまたプログラムなのですから，メモリ空間の一部を占めることは当然です．そこで，ユーザーのマシン語プログラムをどこに置いたらよいかを知るために，N_{88}-BASIC（86）のメモリマップを調べてみましょう（図6-2-1）．

N_{88}-BASIC(86)では，RAMを大きく分けて，システムエリアと4つのセグメント，そしてディスク使用時にはディスクコードエリアの6つに区切り，それぞれ図中に示したような別個の目的のために使用します．

マシン語プログラム・セグメントは，BASIC起動時にはサイズ0であり，セグメント・アドレスmはメモリ使用可能最大アドレスhに等しくなっています．

BASICのCLEAR命令の第2パラメータは，このmの値を変えることによりBASICの使用できるメモリの上限を決め，そのアドレスからRAMの最大アドレスまでをマシン語プログラム用のエリアとして確保します．このようにすることで，マシン語プログラム・セグメント内は，BASICインタープリタの実行とは無関係になり，ユーザーは自由に，BASICとは独立したマシン語プログラムが作成できるのです．

しかし，マシン語プログラム・セグメントのサイズを大きくとりすぎると，BASICの使用する配列データ・セグメントとシンボルテーブル・セグメントが小さくなり，多くの配列や変数を扱

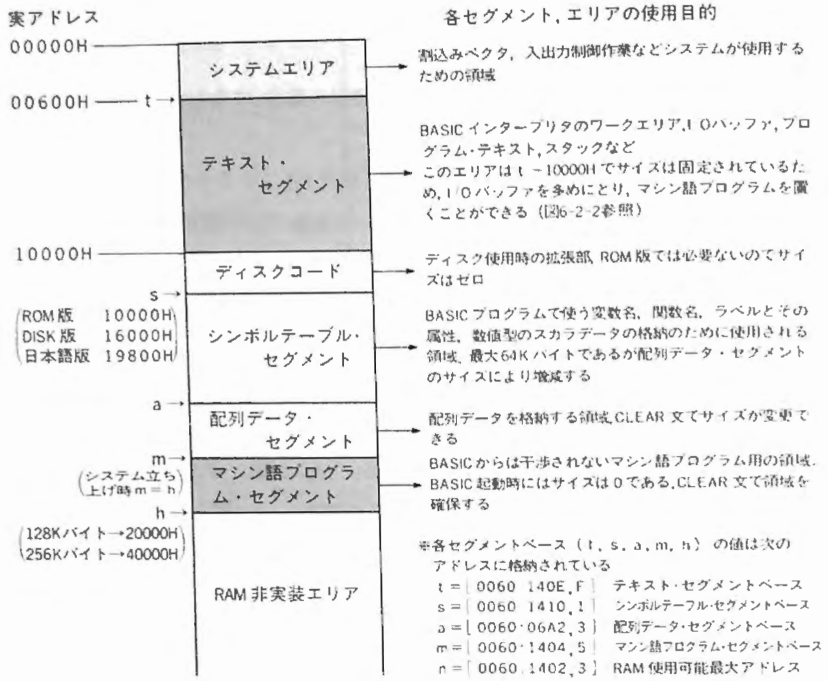


図6-2-1　N88-BASICのRAMメモリマップ(V-RAMを除く)

えなくなるので，注意する必要があります．

この問題を解決するひとつの方法として，マシン語プログラムを通常のマシン語エリアではなく，テキスト・セグメントのI/Oバッファ内におくことが考えられます．

図6-2-1を見ると，テキスト・セグメントのサイズはセグメントベースt(00600H)から10000H番地までで固定されており，立ち上げ時の状態や，ディスクコードのサイズ，CLEAR文などに影響されません．その詳細なマップは図6-2-2のようになっています．

テキスト・セグメントは，下位アドレスから，ワークエリア，I/Oファイルバッファ・エリア，BASICプログラムを記憶するテキストエリア，そしてインタープリタのためのシステム・スタックとユーザー・プログラムのデータ・スタックに分けられています．ワークエリアは，0060：0000番地から0060：1D00番地までで，あとのエリアは，BASIC起動時に"How many files?"で指定する値によりアドレスが決定され，それ以降はシステムをリセットするか，イニシャライズを行うまで変わることはありません．

このI/Oファイルバッファ・エリアは，さらにディスクのメディアに関する情報（5インチか8インチかなど）やデバイスの情報，ファイル管理に必要な情報のためのエリア，そしてI/Oバッファに分かれています．最後のI/Oバッファは，I/O装置とデータの入出力を行う際に，データを一時的にためるための256バイトを単位とした領域で，BASIC起動時に"How many files?(0－

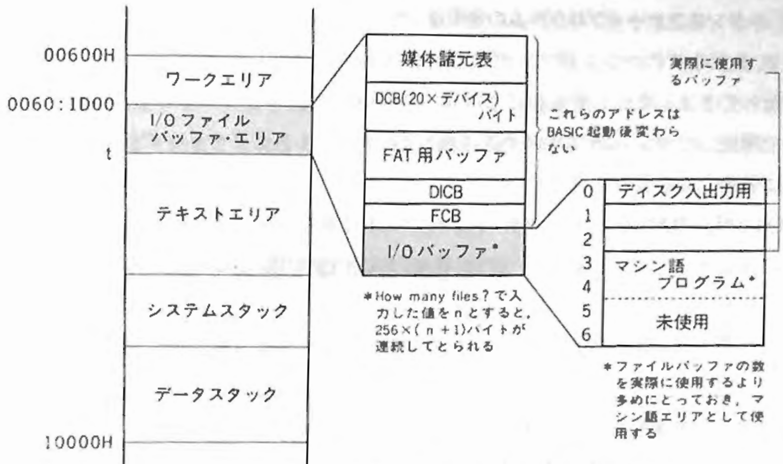


図6-2-2　テキスト・セグメントの詳細

15)" で入力する値をnとすると，

　　256×(n+1)バイト

の領域が連続してとられます．

このことから，I/Oバッファは最大で4Kバイト（n=15）までとれることがわかります．

メモリマップからわかるように，nの値を大きくすると，I/Oバッファがその分連続してとられ，このI/Oバッファは，BASICのOPEN命令で指定しない限り使用されることがありません．

このことは見方を変えれば，BASICプログラムの実行に影響されないメモリエリアがあることになり，ここにマシン語プログラムを書き込めば，配列エリアを減らさずにすみます．

マシン語プログラムをロードするためには，I/Oバッファの先頭アドレスが必要ですが，これは，BASICのVARPTR関数により求められます．

たとえば，I/Oバッファ#4の先頭アドレスは，テキスト・セグメント（0060H）の，

　　オフセット値=VARPTR (#4，0)+33

で求められたアドレスとその次のアドレスに，下位，上位の順で書かれています．

ただし，I/Oバッファ内にマシン語を置くのは，BASICプログラムがあまり大きくなく，同時にOPENするファイルの数が少ない時に適用できる方法です．

グラフィックV-RAMを全く使用しないのであれば，このV-RAM中にマシン語プログラムを置くこともできます．グラフィックV-RAMは，8086のメモリ空間内にあるので，普通のメモリと同様に扱うことができます．また，BASICからはグラフィック命令以外では影響されませんし，SCREEN文でディスプレイページを切り変えれば画面にも影響しません．さらに，マシン語プログラム・セグメントも，むだなI/Oバッファも必要ないため，BASICプログラムに負担をかけません．

6-2-2 トランスファー・プログラムの使用法

ところで、これまでのマシン語プログラミングはすべてMS-DOSの上でデバッガやアセンブラを使って行ってきました．しかし，ここでは，BASIC上でマシン語プログラムを実行することを考えるので開発したマシン語プログラムはMS-DOSのディスクファイルからBASICのディスクファイルに変換する必要があります．

MS-DOSとN_{88}-BASICのディスク・フォーマットは異なっているため、単純にコピーをするというわけにはいきません．これを行うのが巻末のAPPENDIXにのせたトランスファー・プログラムです．

このプログラムは，MS-DOSディスク上のマシン語プログラムをBASICディスクに転送します．

このプログラムを実行すると，MS-DOSのディスクをセットするよう催促されます．ディスクをセットし，そのドライブ番号を入力すると，ディスクのルート・ディレクトリの内容が表示されるので，BASICディスクに転送するファイル名を入力します．ファイル名の指定は大文字でも小文字でもかまいません．

次にBASICディスクに格納する際のファイル名を入力します．すると，

```
Ready (Y/N)
```

と確認を求めますので正しければ"Y"を入力します．それ以外を入力すると再びDOSのファイル名の入力待ちになります．

DOSファイルが見つかると，

```
Find !!
```

と表示され，BASICディスクをセットするよう催促されます．BASICディスクをセットし，そのドライブ番号を入力すると，オフセット値を聞いてきますので，必要なければリターンを入力して下さい．すると,このディスクにマシン語プログラムが書き込まれ，プログラムは終了します．

このプログラムは，転送するマシン語プログラムを，一度，グラフィックV-RAMのB0000H以降にロードしてからBASICのBSAVE命令によってディスクにセーブします．転送できるプログラムのサイズは64Kバイト以内です．

このトランスファー・プログラムで使用可能なMS-DOSディスクの種類（メディア）は次の通りです．

○8インチ2D（両面倍密度）ディスク
○5インチ2DD（両面倍密倍トラック）ディスク
○5インチ2HD（両面高密度）ディスク
○5インチ2D（両面倍密度）ディスク（MS-DOSのシステム・ディスクの場合のみ使用可能）

5インチ2Dディスクで，MS-DOSシステムディスクのみが転送可能な理由は，システムディスクのフォーマットが256バイト／セクタでBASICのディスクフォーマットと同じためです．これは，MS-DOSの標準フォーマット(512バイト／セクタ)とは物理的には異っていますが，BIOSが2セクタを1つにまとめることにより，みかけ上，MS-DOS標準フォーマットと同じにしています．

6-3　BASICからマシン語プログラムを使う

6-3-1　マシン語プログラムのロード

マシン語プログラムを使用する時にまず行うことは，マシン語プログラムをメモリ上に置くことです．その方法としては次の3つが考えられます．

① マシン語プログラムのオブジェクトをDATA文でBASICプログラム中に持たせ，これをPOKE文で1バイトずつメモリに書き込む
② 機械語モニタモードで1行ずつアセンブルするか，メモリに直接16進数で書く
③ 先にマシン語プログラムファイルを作っておき，BLOAD命令で書き込む

①の方法は，BASICプログラムを実行するたびにマシン語プログラムを転送するため時間がかかり，DATA文の分だけテキストエリアが少なくなる欠点はありますが，BASICプログラムとマシン語プログラムの2つが1つのファイルになるという利点があります．この例をリスト6-3-1に示します．

リスト 6-3-1

```
100 ' PROGRAM 6-3-1
110 'Machine lan. loading by POKE statement
120 '
130 CLEAR ,&H1E00
140 MPBASE=&H1E00
150 DEF SEG=MPBASE
160 '
170 EOP=6
180 FOR I=0 TO EOP        'EOP:END OF PROGRAM
190     READ PROG%
200     POKE I,PROG%
210 NEXT I
220 '
230 'PROGRAM
240 DATA &h50             'PUSH  AX
250 DATA &hB4,&h01        'MOV   AH,01H
260 DATA &hCD,&h8A        'INT   08AH
270 DATA &h58             'POP   AX
280 DATA &hCF             'IRET
290 '
300 END
```

②の方法は，比較的短いプログラムの場合に有効です．マシン語プログラム入力後，BSAVE命令でマシン語プログラム・ファイルを作成しておけば，以後は③の方法でマシン語プログラムの読込みができます．

6-1でも説明したように，本章では，MS-DOSのアセンブラでマシン語プログラムを開発し，これを先のトランスファー・プログラムを用いてBASICファイルに変換してBASICとリンクすることを基本に解説しますので，マシン語のロードも③の方法がメインになります．**リスト6-3-2**は，マシン語プログラム・セグメントを確保して，ここにマシン語プログラムをロードするプログラムです．

リスト 6-3-2

```
100 'MACHINE LANGUAGE PROGRAM LOADER 1
110 'Program places on machine lang. program segment.
120 '
130 CLEAR ,&H1E00
140 MPBASE=&H1E00
150 DEF SEG=MPBASE
160 BLOAD "Machine-language-program"
170 '
```

マシン語プログラムをおく領域は，CLEAR文の第2パラメータに，BASICで使用するメモリの上限をセグメント値で指定することで確保します．今，メモリが128Kバイトのシステムを例にすると，

```
CLEAR , &H1E00
```

で実アドレス1E000Hから最終メモリまでの8Kバイトがマシン語プログラムの領域になります．この際，CLEAR文を行う以前に代入された変数はすべてクリアされてしまうので注意してください．実用的に考えれば，CLEAR命令はプログラムの先頭で行う方がよいでしょう．

次にDEF SEG文でマシン語プログラム・セグメントを指定します．これ以後，BASICによるマシン語プログラム・セグメント内の参照（PEEK，POKEなど）は，すべてここからのオフセット値で指定します．

N_{88}-BASIC(86)では，メモリ参照を行う場合，そのアドレスは，DEF SEG文で指定したベース・アドレスと，各メモリ参照命令（PEEK，POKE，BLOAD，BSAVE，CALL，USR等）で指定するオフセット値によって決定されるため，このようなメモリ参照命令の実行前には，DEF SEG文でセグメントベース・アドレスを正しく設定しなくてはなりません．

次にBLOAD命令でマシン語をロードします．これは，

```
BLOAD "ファイル名" ,<ロード・オフセット>
```

の形で，ファイル名で指定されるマシン語プログラム・ファイルを，DEF SEG文で指定したセグメント値に，ロード・オフセットの値を加えた番地からロードします．ロード・オフセットは，指定しなければ0です．

ここで，前の節で説明した，I/Oバッファ内にマシン語をロードする具体例について説明しましょう．リスト6-3-3がそのプログラムです．

リスト 6-3-3

```
100 'MACHINE LANGUAGE PROGRAM LOADER 2
110 'Machine lang. program places on I/O file buffer.
120 '
130 INPUT "ENTER NUMBER OF FILES";FILENUM
140 DEF SEG=&H60
150 OFFSET=VARPTR(#FILENUM,0)
160 MPBASE=VARPTR(#FILENUM,1)
170 IOBUF%=PEEK(OFFSET+32)+PEEK(OFFSET+33)*256
180 PRINT "Program load address is ";
190 PRINT HEX$(MPBASE);":"HEX$(IOBUF%)
200 DEF SEG=MPBASE
210 '
220 BLOAD "Machine-language-program",IOBUF%
```

このリスト6-3-3のプログラムを実行すると，どのI/Oバッファにマシン語をロードするか聞いてきますので，適当な値を入力します．このルーチンを自作のプログラムに組み込むときは，この値はあらかじめ決めておくべきですが，ここではI/Oバッファの先頭アドレスを求める例として，どのファイル番号も指定できるようにしてあります．このとき，起動時に指定した以上のファイル番号を入力しますと，"Bad file number" エラーとなります．

セグメントベースは，テキスト・セグメントなので0060Hとし，VARPTR関数でI/Oバッファの先頭アドレスを得ます．この値がIOBUF%で，このアドレスからマシン語プログラムをBLOAD命令によりロードします．

6-3-2 マシン語サブルーチンコールと引数

BASICプログラムからマシン語プログラムを実行するために，N_{88}-BASICにはCALL文とUSR関数の2つの命令が用意されています．

CALL文の使用法は，まず命令の実行に先立って，任意の変数にマシン語プログラムの開始アドレスをセグメントベースからのオフセット値として代入しておきます．そして，

CALL<変数>

を実行するとマシン語プログラムに制御が渡されます．

USR 関数はまず,

DEF USR<番号>=<開始アドレス>

のように,マシン語プログラムの開始アドレスをマシン語セグメントベースからのオフセット値で指定します.そして,マシン語プログラムの実行は DEF USR で指定した番号を使って,

<変数>=USR<番号>(<引数>)

とします.ここでの引数(ひきすう)は,マシン語プログラムとのデータの受渡しをするための変数のことで,データがない場合でも,ダミーの変数を指定する必要があります.番号には0~9までの数字が許されていますので,10 個までのマシン語プログラムを指定することができ,また,同じ番号でも DEF USR 文で開始アドレスを再設定すれば別のマシン語プログラムを呼び出すことができます.

CALL, USR 命令ともに,その実行前には,マシン語プログラムのセグメントベース・アドレスを DEF SEQ 文で設定しておかなくてはなりません.とくに USR 関数は,DEF USR 文を行ったときのセグメントベース・アドレスを設定します.

リスト 6-3-4 は BASIC プログラムからマシン語プログラムを実行する例で,同じことを CALL 文と USR 関数の2通りで行ってみました.ここで使用したマシン語プログラムは,BASIC の BEEP 1 命令と同じように BEEP 音を出すだけのものです(リスト 6-3-5).

リスト 6-3-4(a) CALL文によるマシン語の実行

```
100 'Sample program of CALL
110 CLEAR ,&H1E00
120 MPBASE=&H1E00
130 DEF SEG=MPBASE
140 MADR=0
150 PRINT "Machine language program begin"
160 CALL MADR
170 PRINT "end"
180 END
```

リスト 6-3-4(b) USR文によるマシン語の実行

```
100 'Sample program of USR
110 CLEAR ,&H1E00
120 MPBASE=&H1E00
130 DEF SEG=MPBASE
140 DEF USR0=0
150 PRINT "Machine language program begin"
160 I=USR0(DUMMY)
170 PRINT "end"
180 END
```

リスト 6-3-5 BEEPルーチン

```
;List 6-3-5
;Program Beep on

main    segment
        assume  cs:main
start:  push    ax      ―― AXの退避
        mov     ax,0    ―― AXのクリア
        mov     ah,1h   ―― AHに1をセット
        int     8ah     ―― BIOSコール
        pop     ax      ―― AXの復帰
        iret    ――┐  ;return to basic
main    ends        └―― BASICへ復帰
        end
```

プログラムの流れは，図6-3-1のようになります．ここで，マシン語プログラムの最後にIRET（Interrupt-on-Return）命令があれば，マシン語プログラムの終了後，呼び出したBASICプログラムへ戻り，次の命令が実行されます．実行に先立ち，リスト6-3-5をアセンブルしたBINファイルをセグメント&H1E00からのメモリにセットしておいてください．

BEEP音はBASICのダイレクト・モードでBEEP 0とすれば止まります．

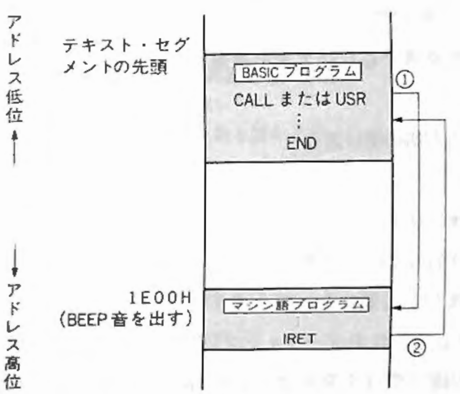


図6-3-1 プログラム実行の流れ

ここで，BASICプログラムからサブルーチンとして呼ばれるマシン語プログラムの注意点をまとめておきます．

マシン語サブルーチンがBASICプログラムへ制御を戻す時の条件は，次の2つがあります．

① BASICインタープリタとマシン語プログラム間のインターフェイス用レジスタBX，AL，および全セグメント・レジスタの値は，マシン語サブルーチンが呼び出された時と同じであること

② SP（スタック・ポインタ）も，呼出し時と同じ値とすること

この条件を満たすために，マシン語サブルーチンの最初で使用するセグメント・レジスタとBX，ALレジスタをスタックへ退避してやります．また，マシン語サブルーチン中でスタックを使用した場合は必ずSPをもとに戻すようPUSHとPOPの数をあわせます．

そしてBASICプログラムへ戻るときには，PUSHしたのと逆の順にレジスタをPOPし，IRET命令で復帰します．

この際PUSH，POP命令は，ワードデータしか扱えないため，ALなどのバイト・レジスタを退避するときはそのワード型であるAXなどを指定してください．

たとえば，これまで作成したマシン語プログラムをBASICからCALLして使用する場合は，

```
PUSH  DS
PUSH  AX
PUSH  BX
    ⋮
マシン語プログラム
    ⋮
POP  BX
POP  AX
POP  DS
IRET
```

のように，プログラム中で使用するレジスタを退避しておくわけです．

6-3-3 共通領域を使った引数の受け渡し

前項で扱ったプログラムは，単にBASICプログラムからマシン語プログラムを実行させるのみの簡単なもので，マシン語プログラムの行う機能もBEEP音を出すだけでした．しかし，マシン語プログラムにもう少し複雑なことを任せようとすると，BASICプログラムとマシン語プログラムの間でのデータの受け渡しが必要になってきます．たとえば，BASICプログラムでデータを用意して，マシン語プログラムでこれをソーティング(並べ換え)するプログラムを考えてみましょう．

まずBASICプログラム側でデータを用意します．次にこれをマシン語プログラムに渡します．マシン語ソートプログラムは，受け取ったデータを昇順または降順に並べ換える処理を行います．そして結果をBASICプログラムへ返します．

このように，サブルーチンを呼び出す親ルーチンがそのサブルーチンに対して入力として渡すデータ，およびそのサブルーチンが親ルーチンに返す結果のデータのことを，総称して引数（ひきすう）というのです．

一般に異なったシステム間，今の場合BASICとマシン語の間で引数を受け渡しする場合には，そのデータ構造を一致させなくてはなりません．これは，異なった人間がコミュニケーションをとるのに同じ言葉を話さなくては通じないのと同じです．

そのための最も一般的な方法は，引数の受け渡しのために両者に共通の領域を作ることです．

そしてBASICプログラムは，マシン語サブルーチンに渡すべき引数をこの領域に書き込んでから，マシン語サブルーチンを呼び出し，呼び出されたマシン語プログラムはこの領域からデータを読み出して処理を行い，再びこの領域に結果を書き込みBASICへ戻します．BASICプログラムはこの領域から処理結果を受け取るわけです．

このときにも，受け渡されるデータの構造，つまり，データがメモリ上に何バイトでどのような順で格納されているか（メモリ内表現），およびそれをどう解釈するか（データ表現）は，同一のものに決めておき，BASICプログラム，マシン語プログラムは，その同一構造のデータを処理するようにする必要があります（図6-3-2）．

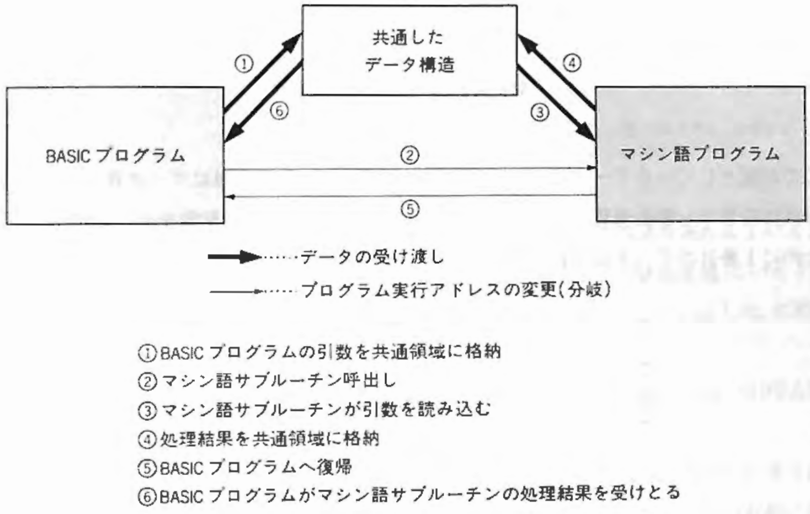


図6-3-2　共通領域による引数の受け渡し

では，これをプログラムで実現してみましょう．

今，BASICプログラムで0~9までの整数データを用意します．このデータは順番がばらばらで，これをマシン語サブルーチンで昇順ソートし，結果をBASICプログラムへ返すことを考えます．

ここで問題となるのは，まずデータ構造をどうするかということです．今扱うのは0~9までの数字であり，これは2進数ならば4ビットで表せます．また文字として扱って，アスキーコードで表すなら30H（0）~39H（9）で1バイトで表すことになります．このようにデータ構造は，1つのデータにしてもいくつもの表現形式が考えられるのです（図6-3-3）．

データ構造は，ソフトウェアにとって最も重要な要素であり，いかに優れたデータ構造を選ぶかによってそのソフトウェアの拡張性，高速性が決定されてしまいます．これはデータ構造が複雑になるほど顕著になります．

データ構造

10進数	2進数	アスキーコード
0	0000	0011 0000 (30H)
1	0001	0011 0001 (31H)
2	0010	0011 0010 (32H)
3	0011	0011 0011 (33H)
4	0100	0011 0100 (34H)
5	0101	0011 0101 (35H)
6	0110	0011 0110 (36H)
7	0111	0011 0111 (37H)
8	1000	0011 1000 (38H)
9	1001	0011 1001 (39H)
	4ビット	8ビット

図6-3-3 いろいろなデータ表現

ここで問題としているデータは，0～9までの整数なので，最も簡単にデータ長は1バイトとし，表現形式は符号なし整数表現とします．また，ソートするデータの個数をマシン語プログラムに渡すために1番目のデータにこれを設定します（図6-3-4）．

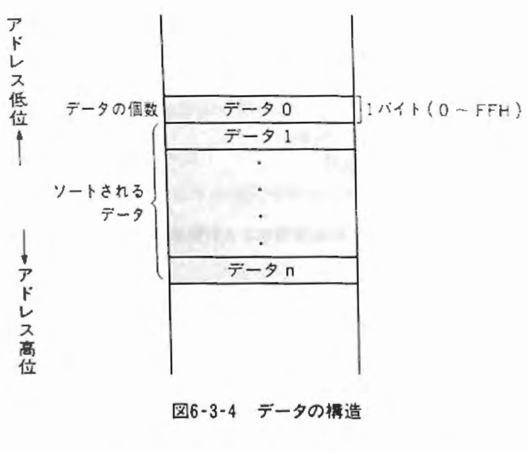


図6-3-4 データの構造

BASIC側のプログラムをリスト6-3-6に，マシン語側のプログラムをリスト6-3-7に示します．

BASICプログラムでは，まずマシン語プログラムをロードします．このときマシン語プログラムのセグメントベースは1E00Hからとし，プログラムをオフセット0から，また引数の受渡しのための共通領域はセグメント値1F00Hからとします．そしてソートするデータの個数をオフセット0に，データをその後に連続してセットします．これはPOKE文を使っていますが，POKE文の前には，この共通領域のセグメントベースをDEF SEG文でセットすることを忘れないように

します．ここでは，共通領域は固定したアドレス（1F00H）から始まることにしており，プログラム・セグメントとデータ・セグメントははっきり分けています（図6-3-5）．

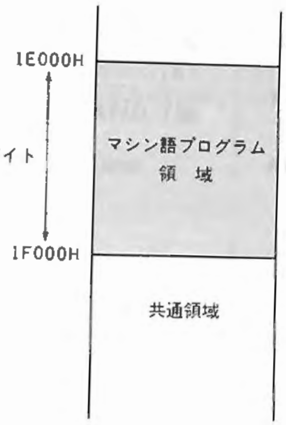


図6-3-5　マシン語プログラム・セグメントのメモリマップ

BASICプログラムがマシン語プログラムに渡す引数は，BASICの内部表現から1バイトのデータ構造に変換しなくてはなりませんが，幸いPOKE文は，BASICの整数表現(2バイト)で表される0～255までの整数を，1バイトにしてメモリに書き込むようになっているため，BASICプログラムではとくに複雑な変換プログラムは必要ありません．

データを共通領域に準備した後には，セグメントをマシン語プログラムの領域(MPBASE)に設定し直してCALL文でサブルーチン・コールします．

マシン語サブルーチンから戻ったら，ソートされたデータをPEEK関数で読み，結果を出力します．PEEK関数も，1バイトのデータを対応する0～255のBASIC整数表現に自動的に変換してくれます．

一方，マシン語プログラムは，共通領域のセグメントベース1F00HをDS（データ・セグメント）レジスタに入れ，ソーティングを行います．結果は，BASICが共通領域に書いたのと同じ位置に格納します．ここでは，ソート・ルーチンのアルゴリズムには以前使用したバブルソートと呼ばれるものを使用しました．

リスト 6-3-6 BASIC部（共通領域用）

```
100 ' PROGRAM 6-3-6
110 ' DATA PREPEAR
120 DEFINT A-Z
130 'MACHINE LAN.PROGRAM LOADER
140 CLEAR ,&H1E00
150 MPBASE=&H1E00
160 DBASE=&H1F00
170 DEF SEG=MPBASE
```

```
180 BLOAD "L6-3-7"
190 '
200 DATA 10                              ' NUMBER OF DATA
210 DATA 4,9,1,6,2,5,8,3,7,0             ' DATA FOR SORT
220 '
230 DEF SEG=DBASE
240 READ N.DATA                          ' SET NUMBER OF DATA
250 POKE 0,N.DATA
260 PRINT "DATA BEFORE SORT  : ";
270 FOR I=1 TO 10                        ' SET DATA FOR SORT
280      READ BDATA
290      PRINT BDATA;:POKE I,BDATA
300 NEXT I:PRINT
310 '
320 DEF SEG=MPBASE
330 MADR=&H0
340 CALL MADR
350 ' GET SORTED DATA
360 DEF SEG=DBASE
370 PRINT "DATA AFTER SORTED : ";
380 FOR I=1 TO 10
390      PRINT PEEK(I);
400 NEXT I
410 '
420 END
```

リスト 6-3-7 バブルソートルーチン（共通領域用）

```
;List 6-3-7
;Program of bubble sort for List 6-3-6
;                   (Byte data)

dbase   equ     1f00h   ――――――OSにセットする値

main    segment
        assume  cs:main,ds:main
begin:
        push    ds      ┐
        push    ax      │
        push    bx      │ 使用するレジスタの退避
        push    cx      │
        push    si      │
        push    di      ┘
        mov     ax,dbase ┐ DSに1F00Hをセット
        mov     ds,ax    ┘
        mov     bx,0h    ┐
        xor     ax,ax    │
        mov     al,[bx]  │
        mov     di,ax    │
        inc     bx       │
out_loop:                │
        dec     di       │
        cmp     di,1h    │
        jb      e_prog   │
        mov     cx,di    │ 1バイトデータのバブルソート
        dec     cx       │ DS：BX＝データ領域のアドレス
        mov     si,0     │ DI＝データの数n
in_loop:
```

```
        cmp     si,cx
        ja      out_loop
        mov     al,[si][bx]
        mov     ah,1[si][bx]
        cmp     al,ah
        jbe     e_swap
        xchg    al,ah            データの入れ替え
        mov     [si][bx],al
        mov     1[si][bx],ah
e_swap:
        inc     si
        jmp     in_loop
e_prog:
        pop     di
        pop     si
        pop     cx               レジスタの復帰
        pop     bx
        pop     ax
        pop     ds
        iret                     ;return to basic

main    ends
        end
```

6-3-4 CALL文による引数の受け渡し

共通領域を作ってBASICプログラムとマシン語プログラムの間で引数の受渡しをする方法は，とくにBASICに限らず，他の言語についても適用できる一般的なものです．しかし，これには，BASICのデータ構造を別のデータ構造に変換し，共通領域に転送し，マシン語側でもこれにあわせて処理を行い，結果をまたBASICのデータ構造に直すという手間がかかり，マシン語プログラムとリンクしても，この変換と転送に時間を食われて全体の実行速度が遅くなってしまうという欠点があります．

そこで，考えられるのが引数のデータ構造をBASICの内部表現に合わせてしまうことです．こうすればBASICの内部表現から他のデータ構造への変換の必要はなくなります．

さらにもう一歩進んで引数受渡しの共通領域をなくし，マシン語プログラムが直接BASICの変数領域を参照するようにすれば，共通領域へのデータ転送もなくなり，かなりの高速化が期待できます．

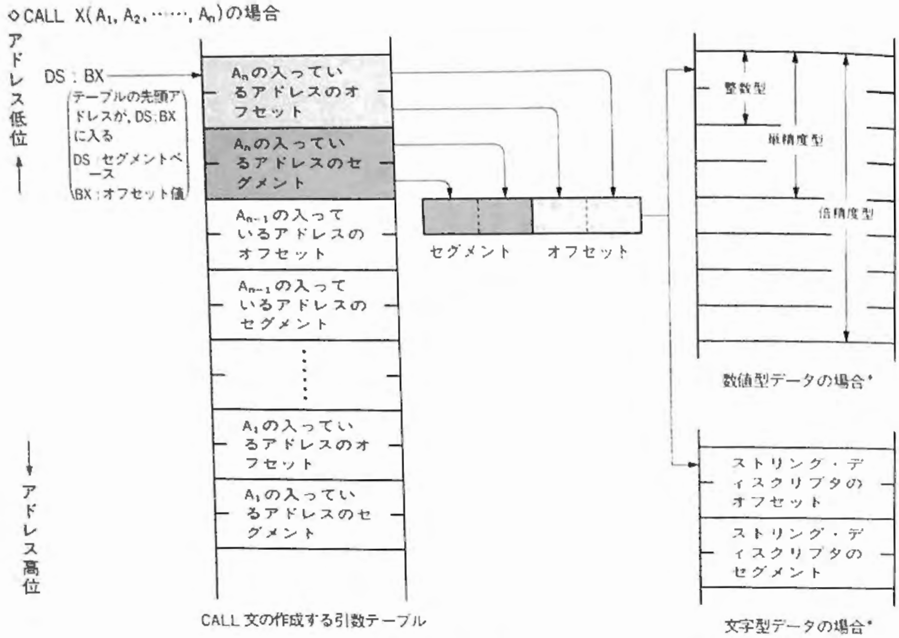
マシン語プログラムがBASICの変数を直接参照するためにN_{88}-BASICではCALL文とUSR関数に引数を渡す機能が用意されています．

CALL文による引数の受渡しの例を解説しましょう（図6-3-6）．

CALL文によって引数の受渡しをするには次のようにします．

CALL<変数名>(引数1，引数2，……，引数n)

<変数名>には前述のようにマシン語プログラムの開始アドレスのオフセット値が入ります．



＊データの内部表現については「ユーザーズ・マニュアル19.9変数のメモリ表現」を参照

図6-3-6　CALL 文の引数の受け渡し

引数は，このように複数指定することができます．

CALL 文での引数の受渡しは，それぞれの引数に対応した変数のデータが格納されているメモリの先頭アドレスを順に並べたテーブルにより行います．このテーブルは，図6-3-6のようにアドレスの低い方から引数n，n−1，……，の順でデータのアドレスのセグメントベース，オフセット値が入っています．このアドレスは，BASIC の VARPTR 関数で得られるのと同じアドレスです．そして CALL 文が実行されるとこのテーブルの先頭アドレスのセグメント値とオフセット値がそれぞれ DS，BX レジスタにセットされ，マシン語プログラムに渡されます．

マシン語プログラムでは，このテーブルから各引数のデータのアドレスを得てデータを参照します．

引数が文字型の場合にはテーブルに書かれるアドレスはストリング・ディスクリプタのアドレスになります．

CALL 文で呼び出されたマシン語サブルーチンは，このアドレスを介して BASIC プログラムに値を返すことができます．これは引数のデータ自身を書き変えることであり，BASIC プログラムに戻ったときにはその引数の変数の内容が変わることになりますので，BASIC プログラムではその変数の内容をそのまま使用することができます．ただし，データの型を変えることはできません．

CALL 文による引数の受渡しで気をつけなくてはならないことは，引数の型とその個数はマシン語に渡されないことです．したがって，マシン語プログラムを作るときには，引数の順番と型

はあらかじめ決めておき，BASICプログラムと整合させなくてはなりません．また引数の個数も合わせておくか，個数自体を引数の１つとして渡してやることが必要です．

ここで，前出の共通領域を通して引数を受け渡したソート・プログラムの例を，BASICの整数型データ構造を用いてマシン語プログラムから直接に変数のデータを参照する方法で作ってみます．

リスト6-3-8はBASIC部分のプログラム，リスト6-3-9はマシン語によるソート・ルーチンです．

リスト 6-3-8　BASIC部(CALLの引数用)

```
100 ' PROGRAM 6-3-8
110 'MACHINE LAN.PROGRAM LOADER
120 CLEAR ,&H1E00
130 DEFINT A-Z
140 MPBASE=&H1E00
150 DEF SEG=MPBASE
160 BLOAD "L6-3-9"
170 '
180 ' DATA PREPEAR
190 N.DATA=10                                    ' NUMBER OF DATA
200 A=4:B=9:C=1:D=6:E=2:F=5:G=8:H=3:I=7:J=0 ' DATA FOR SORT
210 '
220 PRINT "DATA BEFORE SORT  : ";A;B;C;D;E;F;G;H;I;J
230 '
240 MADR=&H0
250 CALL MADR(A,B,C,D,E,F,G,H,I,J,N.DATA)
260 '
270 ' PRINT SORTED DATA
280 '
290 PRINT "DATA AFTER SORTED : ";A;B;C;D;E;F;G;H;I;J
300 END
```

リスト 6-3-9　バブルソートルーチン(CALLの引数用)

```
;List 6-3-9
;Program of bubble sort for List 6-3-8

wkofset equ     1000h

main    segment
        assume  cs:main,ds:main
begin:
        push    ds          ┐
        push    es          │
        push    ax          │
        push    bx          │ レジスタの退避
        push    cx          │
        push    si          │
        push    di          ┘

        les     si,[bx] ──── CALL文の引数テーブルからN.DATAのアドレスを
        mov     cx,es:[si]   ES：SIにロードする
```

```
        push    cx          ──── データの数をPUSHしておく
lddata:
        mov     di,wkofset
loop1:
        add     bx,4
        les     si,[bx]
        mov     ax,es:[si]
        mov     cs:[di],ax      引数の値(ソートするデータ)をワークエリア(CS：1000)へ転送する
        inc     di
        inc     di
        loop    loop1
;
        pop     di
        push    di              データ数,テーブルのポインタ・アドレスのセグメントベースと
        push    ds              オフセット値の保存
        push    bx
;
        mov     ax,cs
        mov     ds,ax           CS=DSにセット
        mov     bx,wkofset
        add     di,di       ──── ワードデータのソートなのでnを2倍にする
out_loop:
        dec     di
        dec     di
        cmp     di,2h
        jb      e_prog
        mov     cx,di
        dec     cx
        dec     cx
        mov     si,0
 in_loop:
        cmp     si,cx
        ja      out_loop        ワークエリア上でのワードデータのバブルソート
        mov     ax,[si][bx]
        mov     dx,2[si][bx]
        cmp     ax,dx
        jbe     e_swap
        xchg    ax,dx
        mov     [si][bx],ax
        mov     2[si][bx],dx
  e_swap:
        inc     si
        inc     si
        jmp     in_loop
  e_prog:
  ;
        pop     bx              データ数,テーブルのポインタ・アドレスのセグメントベース
        pop     ds              およびオフセット値の復帰
        pop     cx
        mov     si,wkofset
  loop2:
        mov     ax,cs:[si]
        les     di,[bx]
        mov     es:[di],ax
        sub     bx,4            ソートしたデータをメモリに戻す
        inc     si
        inc     si
        loop    loop2
  ;
        pop     di
```

```
        pop     si
        pop     cx
        pop     bx          レジスタの復帰
        pop     ax
        pop     es
        pop     ds
        iret                ;return to basic
;
main    ends

        end
```

BASICプログラムでは，マシン語プログラムのロード後，ソートする10個のデータを変数A～Jに代入し，画面に表示したのち，すべてをCALL文の引数にします．また，引数の個数をN．DATAに代入し，最後の引数にします．

マシン語ルーチンによるソート終了後は，同じ変数A～Jにソートした結果が格納されているのでこれをプリントします．

一方，マシン語プログラムは，まず使用するレジスタをPUSH命令でスタックへ退避します．次にDS，BXレジスタに入っている引数のアドレス・テーブルの先頭アドレスを用いて，各引数の値を参照します．このとき，BXが指しているのは引数N．DATAの値が格納されているアドレスの格納されているアドレスであり，この値を参照することでマシン語プログラムは最初にソートすべきデータの値を知ることができます．

次にオフセット1000H番地からをワーク・エリアとし，ここに引数A～Jのデータを転送します．そして，ソーティングを行っ後，結果をもとのアドレスに返します．最後に使用したレジスタをPOP命令でもとに戻し，IRET命令でBASICに復帰します．

ところで今のプログラムではソートするデータが10個だったので，1つずつ変数に代入してCALL文の引数にしました．しかし，データ数がもっと多くなった場合，代入するだけでも大変です．

BASICプログラムでのこのような数の多いデータの処理には，配列変数を使用するのが一般的です．この配列変数をマシン語の引数とする方法を考えてみましょう．

図6-3-7は整数型1次元の配列データの格納の様子です．

このように整数型配列データは，配列データ・セグメントの中に添字0の変数から順に連続して格納されており，これは単精度型や倍精度型でも同様です．

これを利用すれば，CALL文の引数としてテーブルの先頭にある添字0の配列変数のアドレスと，配列データの個数をマシン語プログラムに渡すことで，多数のデータの処理が可能となります．前のプログラムならデータを配列D%に代入して，

```
CALL MADR(D%(0), N . DATA)
```

とします．

配列では，全データが順序よく連続したアドレスにあることから，マシン語プログラム側では，ワーク・エリアをとることなく配列データ自体をソートすることができます．このように改良したプログラムをリスト 6-3-10，6-3-11 に示します．

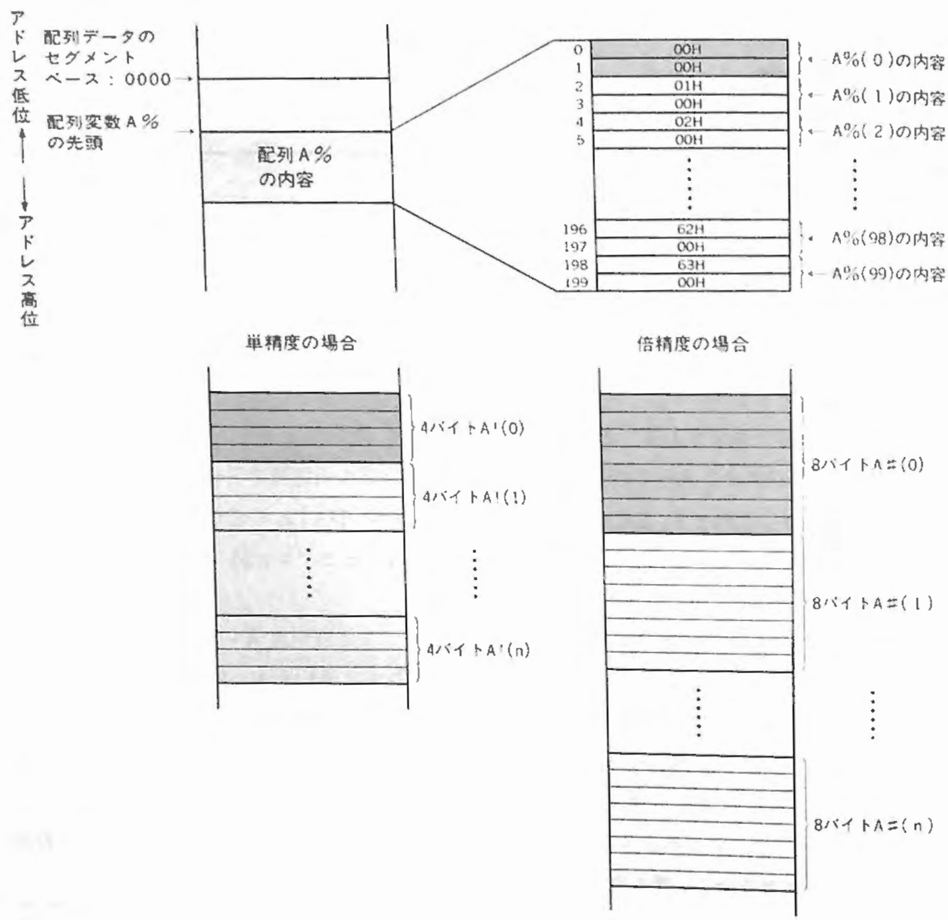


図 6-3-7　配列データの格納状態

リスト 6-3-10 BASIC部(配列データ用)

```
100 ' PROGRAM 6-3-10
110 'MACHINE LAN.PROGRAM LOADER
120 CLEAR ,&H1E00
130 DEFINT A-Z
140 MPBASE=&H1E00 : DBASE=&H1F00
150 DEF SEG=MPBASE
160 BLOAD "L6-3-1.1"
170 DATA 10                                ' NUMBER OF DATA
```

```
180 DATA 4,9,1,6,2,5,8,3,7,0          ' DATA FOR SORT
190 '
200 ' DATA PREPEAR
210 READ N.DATA:DIM D(N.DATA)         ' SET NUMBER OF DATA
220 PRINT "DATA BEFORE SORT  : ";
230 FOR I=0 TO N.DATA-1               ' SET DATA FOR SORT
240     READ BDATA:D(I)=BDATA
250     PRINT BDATA;
260 NEXT I:PRINT
270 '
280 DEF SEG=MPBASE
290 MADR=&H0
300 CALL MADR(D(0),N.DATA)
310 '
320 ' GET SORTED DATA
330 DEF SEG=DBASE
340 PRINT "DATA AFTER SORTED : ";
350 FOR I=0 TO N.DATA-1
360     PRINT D(I);                   ' PRINT SORTED DATA
370 NEXT I
380 '
390 END
```

リスト 6-3-11 バブルソートルーチン(配列データ用)

```
; List 6-3-11
; Program of bubble sort for List 6-3-10

main    segment
        assume  cs:main,ds:main
begin:
        push    ds              ┐
        push    es              │
        push    ax              │
        push    bx              │レジスタの退避
        push    cx              │
        push    si              │
        push    di              ┘
;
        les     si,[bx]─────────CALL 文の引数N.DATA の値を得る
        mov     di,es:[si]         ;di has num_data
        les     bx,4[bx]────────配列変数の先頭アドレスを得る
        mov     ax,es           ┐DS=ES にセット
        mov     ds,ax           ┘
;
        add     di,di                 ; for word data
out_loop:
        dec     di              ┐
        dec     di              │
        cmp     di,2h           │
        jb      e_prog          │
        mov     cx,di           │
        dec     cx              │
        dec     cx              │
        mov     si,0            │配列変数のデータそのものをソートする
in_loop:                        │
        cmp     si,cx           │
        ja      out_loop        │
```

```
        mov     ax,[si][bx]
        mov     dx,2[si][bx]
        cmp     ax,dx
        jbe     e_swap
        xchg    ax,dx
        mov     [si][bx],ax
        mov     2[si][bx],dx
e_swap:
        inc     si
        inc     si
        jmp     in_loop
e_prog:
        pop     di
        pop     si
        pop     cx
        pop     bx              レジスタの復帰
        pop     ax
        pop     es
        pop     ds
        iret                            ;return to basic

main    ends

        end
```

6-4　BASIC-ROMの利用

6-4-1　BASIC-ROMルーチンとは

これまでのサンプル・プログラムで，マシン語プログラムによって処理したデータはすべて整数型でした．これは，マシン語が整数型データを扱うのに向いている反面，実数型データ構造は扱いにくくプログラムが極端に複雑になるからです．たとえば単精度型データ同士の減算を考えた場合，まず両方のデータの指数部をあわせ，それに伴って仮数部も変換する必要があります．これをマシン語によって行うには複雑なプログラムが必要になります．

このような実数型データの演算をマシン語で行いたいという場合にBASICのROMルーチンを使用することができます．PC-9801のROM内には図6-4-1に示すようないくつかのシステム・プログラムが納められており，BASICインタープリタもこのうちの1つです．実数型データ演算や組込み関数も，その中にマシン語サブルーチンの形で用意されており，BASICインタープリタはそのような計算を行うときに，必要に応じてこれらを呼び出して使用しているのです．そこでこれらのサブルーチンをユーザーのマシン語プログラムから呼び出せればマシン語プログラムで実数型データの演算を扱うことができます．

ROMの内部には，BASICインタープリタ，ターミナルモード・プログラム，機械語モニタ・プログラムなどのソフトウェアの他，グラフィック制御の基本ルーチン(BIO)や論理的なインターフェイス・ルーチン（LIO），CRTやキーボードの制御，ハードコピー・ルーチン，基本的な入

出力を行うルーチン (BIOS) などが含まれています．これらのプログラムのソフトウェア的な関係は図 6-4-2 のようになっています．

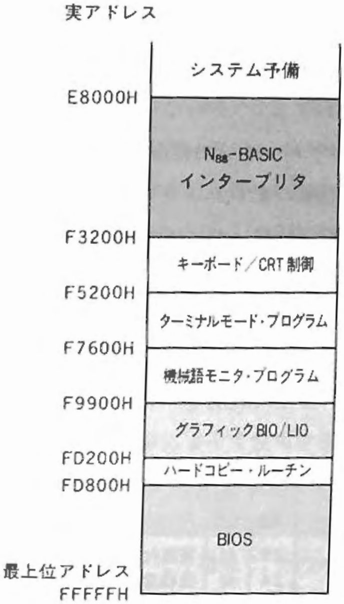


図 6-4-1　BASIC-ROM メモリマップ

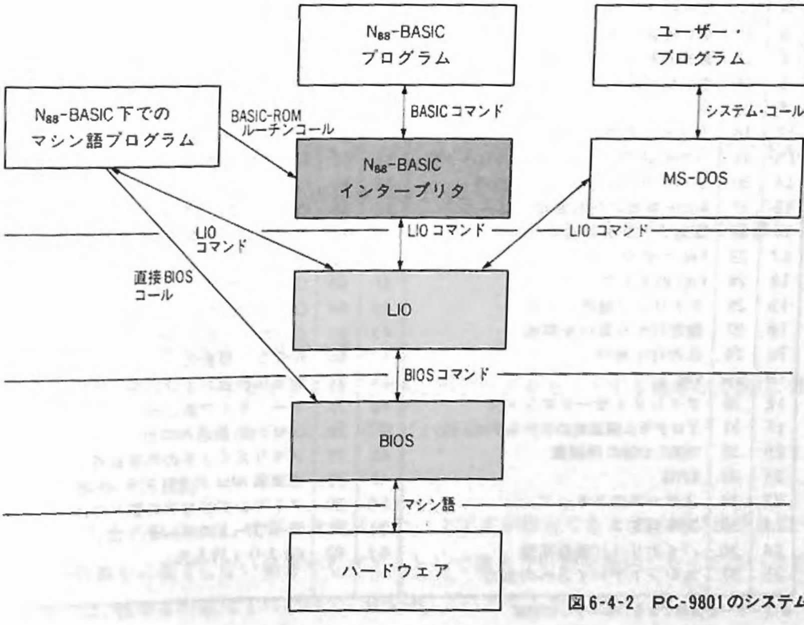


図 6-4-2　PC-9801 のシステム構造

ハードウェアを直接制御しているのはBIOSであり，BASICインタープリタはLIOを通してBIOSを利用することで，プリンタやディスクなどのI/O装置とデータのやりとりを行っています．BASICインタープリタ内のマシン語サブルーチンには先に述べた演算のプログラムの他に，BIOSを使用するプログラムも用意されています．

6-4-2 BASIC-ROMルーチンの呼び方

BASIC-ROMのうちインタープリタ内のサブルーチンの使用には，一般にソフトウェア割込みINT C4Hを使用するディスパッチャ割込みを用います．

この方法で使用できるルーチンには，すべてルーチンナンバーが付いており，その番号をDIレジスタにセットしてから，INT C4Hを実行することで自動的にROM内のテーブルが参照され，ルーチンナンバーに該当するマシン語サブルーチンが実行されます．これはROMルーチン・ディスパッチャと呼ばれるルーチンにより行われます．この方法で使用できるルーチンのナンバーと機能の一覧表を表6-4-1に示します．

表6-4-1 BASIC-ROMルーチン一覧

ルーチンNo. 16進	10進	機能	ルーチンNo. 16進	10進	機能
5	5	倍精度加算	29	41	実数化(CSNG)
6	6	倍精度乗算	2A	42	倍精度化(CDBL)
7	7	倍精度除算	2B	43	単精度加算
8	8	倍精度の整数化(INT)	2C	44	単精度減算
9	9	倍精度化(CDBL)	2D	45	単精度乗算
A	10	オーバーフロー処理	2E	46	単精度除算
B	11	ゼロ除算処理	2F	47	単精度比較
C	12	数値の整数化(CINT)	30	48	倍精度減算
E	14	数式評価	31	49	倍精度比較
F	15	ストリング・ディスクリプタの生成	32	50	バイナリ→8進数変換
12	18	ファイル番号の評価，チェック	33	51	バイナリ→16進数変換
13	19	ファイル・ディスクリプタのコード化	34	52	符号なし整数値→10進数変換
14	20	デバイスタイプコードの取得	3B	59	CRTへの表示(条件付)
15	21	A00H番地の内容番地をコール	3C	60	CRTへの表示(無条件)
16	22	変数アドレスの取得	3D	61	1文字CRTへ表示
17	23	FACへのロード	3E	62	カーソルのリセット
18	24	FACのストア	3F	63	CR, LFの出力
19	25	ストリング値のロード	40	64	CRTへCR, LF出力
1B	27	指定行から実行を開始	41	65	CRTカーソルのリセット
1C	28	次の行を実行	42	66	符号なし整数化
1D	29	VAL	4A	74	数式の評価および結果の整数化
1E	30	ダイレクトモードエントリ	4B	75	キー，タイマ割込みのセンス
1F	31	プログラム編集時のステータスリセット	4C	76	COM, PEN割込みのセンス
20	32	PRINT USING用編集	4E	78	メモリスイッチの内容取得
21	33	STR$	4F	79	未実装RAMアクセスチェック
22	34	スペースのスキップ	50	80	ストリングエリアの割りつけ
23	35	文末判定	51	81	キーワードのサーチ
24	36	バイナリ→10進数変換	52	82	KBより1行入力
25	37	カレントデバイスへの出力			

＊ユーザーが使用できないルーチンは削除

BASICインタープリタ内のサブルーチンを使用するもう1つの方法に，ユーザーのマシン語プログラムから，インタープリタ内のサブルーチンをFAR CALL命令によって直接呼び出すものがあります．

しかしながら，この方法では，呼び出すサブルーチンの開始アドレスを知る必要があり，PC-9801ではROMのいくつかのバージョンによりROM内ルーチンの絶対アドレスが違っているため，すべての機械で使用できる保障がないのでここでは割愛することにします．

これに対して，先のROMルーチン・ディスパッチャによる方法は，ユーザーがPC-9801のROMを利用できるようにするために取り入れられた機能であり，バージョンによるROM内アドレスの違いがあっても，変わることがないように保障されているので，すべてのPC-9801で使用することができます．

6-4-3　ディスパッチャの使用法

ここでは，ディスパッチャを用いたROM内ルーチンの使用法を解説しましょう．

ROMルーチン・ディスパッチャによって使用できる数値演算ルーチンには単精度，倍精度の四則演算および比較があり，これらはすべて浮動小数点アキュムレータ(Floating-point Accumulator：FAC）と呼ばれるBASIC内部の仮想レジスタ上で行われます．FACは実際にはテキスト・セグメント内に存在する8バイトのメモリ領域で，そのオフセット・アドレスは，1416～141DH番地です．FAC内のデータの型は，オフセット1414Hに数値でセットされ，その意味は次の通りです．

[1414H]＝
- 2：整数型
- 3：文字型
- 4：単精度実数型
- 8：倍精度実数型

また，四則演算，比較の場合は，レジスタがもう1つ必要なため1420H～1427HにFAC2が用意されます．FACとFAC2の間での演算結果はFACに入ります．

次頁の**表6-4-2**にROMディスパッチャルーチンにより用いる実数型データの演算とFACに関するルーチンをまとめておきます．

6-4-4　USR関数を使ったデータの受け渡し

実数型データ演算ルーチンを使用する時に，FACへデータをセットする簡単な方法に，BASICのUSR関数を使う方法があります．USR関数は，前述のように，

USR<番号>(引数)

の形で使用し，引数の部分には，任意の数値式または文字式が指定できます．呼び出されたマシン語ルーチンが引数を必要としない場合でもダミーとして適当な引数を指定しなくてはいけません．

USR関数では，渡せる引数は1つだけで，BASICプログラムからマシン語プログラムへの引数

表6-4-2 実数型データ演算およびFACに関するROM内ディスパッチャルーチン(INT C4H)

DIにセットする値	機　能	結　　果
5H	倍精度加算	FAC＝FAC2＋FAC
6H	倍精度乗算	FAC＝FAC2＊FAC
7H	倍精度除算	FAC＝FAC2/FAC
8H	倍精度整数化	FAC＝CINT(FAC)
9H	倍精度化	FAC＝CDBL(FAC)
29H	単精度化	FAC＝CSNG(FAC)
2AH	倍精度化	FAC＝CDBL(FAC)
2BH	単精度加算	FAC＝FAC2＋FAC
2CH	単精度減算	FAC＝FAC2－FAC
2DH	単精度乗算	FAC＝FAC2＊FAC
2EH	単精度除算	FAC＝FAC2/FAC
2FH	単精度比較	結果は整数型でFACにはいる FAC＞FAC2, FAC＝FAC2, FAC＜FAC2 に応じて1, 0, －1
30H	倍精度減算	FAC＝FAC2－FAC
31H	倍精度比較	結果は整数型でFACにはいる FAC＞FAC2, FAC＝FAC2, FAC＜FAC2 に応じて1, 0, －1

＊表の値をDIレジスタに入れたのち，INT C4を行う

およびマシン語プログラムの処理結果の受渡しには，FACが使用されます．

USR関数が実行されると，ALレジスタにはマシン語プログラムへ渡された引数の型を示す値がセットされます．また，DSレジスタにはシンボル・テーブルのセグメントベース・アドレスがセットされます．シンボルテーブル・セグメントには，変数やストリング・ディスクリプタが格納されています．またBXレジスタには，FAC内のデータの先頭アドレスを示すDSレジスタの値をセグメントベースにするオフセット値が入っています．

USR関数実行時にマシン語プログラムへ渡される各レジスタ情報を表6-4-3にまとめます．

表6-4-3 USR関数実行時のレジスタの情報

AL	2：整数型
	3：文字型
	4：単精度実数型
	8：倍精度実数型
DS	シンボルテーブルのセグメントベース
BX	FACのオフセット・アドレス
DX	ストリング格納領域のセグメントベース(文字型のみ)

FACの内容は引数の型によって図6-4-3のようになります。

数値型データではFAC内にデータ自身が格納されていますが，文字型の場合は，BXレジスタがFACの5バイト目を指し，そこに，ストリング・ディスクリプタの先頭アドレスがセグメントベースとオフセットの形で入っています．実際の文字列データは，さらにストリング・ディスクリプタの2バイト目のリロケーションコードの値により文字列データの格納されているセグメントベースを決定し，3バイト目，4バイト目をオフセット値としてそのアドレスが得られます．

リロケーションコードが0ならば文字列データのセグメントはDXレジスタの値であり，0でないときは0060Hつまりテキスト・セグメントがセグメントベースになります．

USR関数により呼び出したマシン語サブルーチンが，BASICプログラムへ結果を返すには，デ

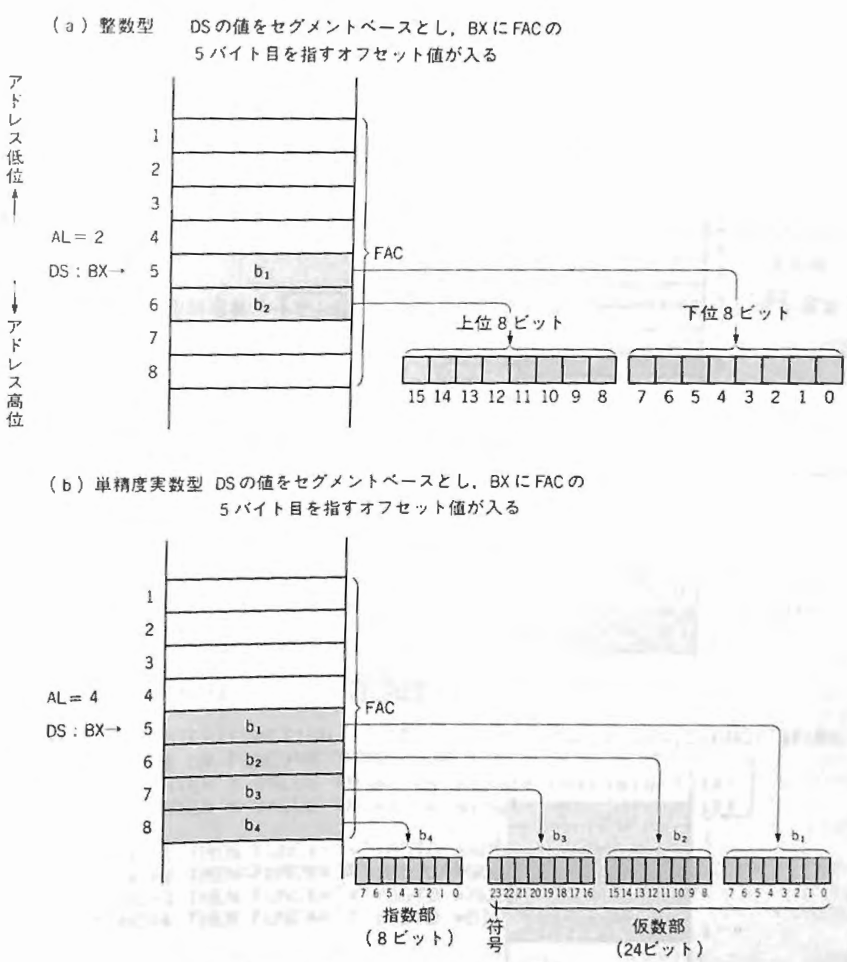
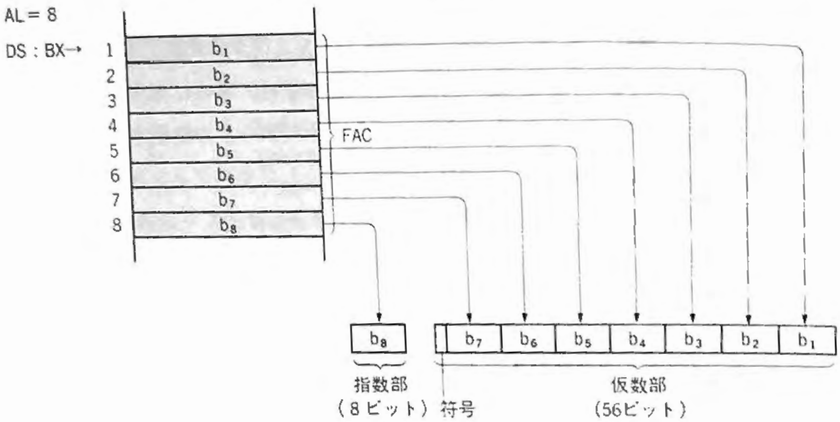


図6-4-3 FAC内のデータ表現

（c）倍精度実数型　DSの値をセグメントベースとし，BXにFACの1バイト目を指すオフセット値が入る



（d）文字型　DSの値をセグメントベースとし，BXにFACの5バイト目を指すオフセット値が入る

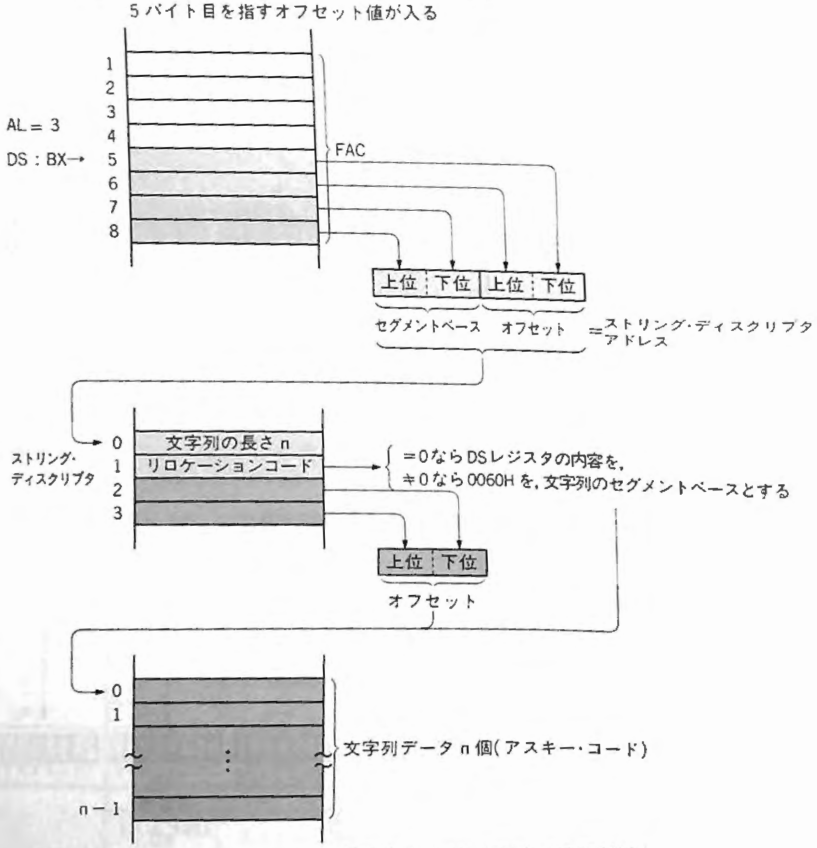


図6-4-3　FAC内のデータ表現（続き）

ータをFACにセットしてALレジスタに返すデータの型に対応した数値をセットします.返すデータは，引数として渡された値と同じ型でなくてはなりません.

6-4-5　ROM内演算ルーチンを使った実数計算プログラム

ここでは，ROM内の実数型データ演算ルーチンを呼び出すプログラムを紹介します.

リスト6-4-1は，USR関数を使って，単精度型の実数データの受け渡しと四則演算をするプログラムです.

リスト6-4-2のマシン語ルーチンをセグメント1E00Hの0000H番地から書いておき，X, Yに実数値を代入したのち，必要なルーチンをコールします.

マシン語ルーチンは，5つの部分からなり，0~21H番地までは，FACの内容とFAC2に転送するルーチンです．以降，22H番地から27H番地までがBASIC-ROM内の単精度型データ用の加算ルーチンを呼び出すベクタ(2BH)を代入する部分，28H番地から2DH番地まで加減算ルーチンを呼び出すベクタ(2CH)を代入する部分，2EH番地から33H番地までが，乗算ルーチンを呼び出すベクタ(2DH)を代入する部分，最後の34H番地から38H番地が除算ルーチンを呼び出すベクタ(2FH)を代入する部分です.

これらのルーチンは，BASICリストのUSR0~4に対応しており，そのUSR関数を指定してコールするわけです.

また，ROM内の四則演算ルーチンは，すべてFAC2の内容に対してFACの内容を演算する形で行われ，結果がFACに格納されるので，この値をBASICリストの300行~330行のRESULTに代入し，出力しています（リスト6-4-1，6-4-2).

リスト 6-4-1　実数計算BASIC部

```
100 ' USR sample program
110 '    and INT C4H sample program
120 CLEAR ,&H1E00:DEF SEG=&H1E00
130 BLOAD "L6-4-2"
140 DEF USR0=0
150 DEF USR1=&H22
160 DEF USR2=&H28
170 DEF USR3=&H2E
180 DEF USR4=&H34
190 '
200 INPUT "ENTER FUNCTION (1:ADD,2:SUB,3:MUL,4:DIV,5:END)";FUNC
210 IF FUNC<0 OR FUNC>=5 THEN *EOP
220 INPUT "ENTER X VALUE (Must be single precision)";X!
230 INPUT "ENTER Y VALUE (Must be single precision)";Y!
240 '
250 IF FUNC=1 THEN FUNC$="+":GOTO *ADD
260 IF FUNC=2 THEN FUNC$="-":GOTO *SUB
270 IF FUNC=3 THEN FUNC$="*":GOTO *MUL
280 IF FUNC=4 THEN FUNC$="/":GOTO *DIV
290 '
```

```
300 *ADD:DUMMY!=USR0(X!):RESULT!=USR1(Y!):GOTO 340
310 *SUB:DUMMY!=USR0(X!):RESULT!=USR2(Y!):GOTO 340
320 *MUL:DUMMY!=USR0(X!):RESULT!=USR3(Y!):GOTO 340
330 *DIV:DUMMY!=USR0(X!):RESULT!=USR4(Y!):GOTO 340
340 PRINT X!;FUNC$;Y!;"=";RESULT!
350 '
360 GOTO 200
370 *EOP:END
```

リスト 6-4-2 実数計算マシン語ルーチン

```
        ; List 6-4-2
        ; Program of real single precision calculation
        ;                          for List 6-4-1
        fac_mem         segment  at 60h        ——FAC, FAC2 のセグメント
                        org      1416h
        fac             dw       4 dup (?)

                        org      1420h
        fac2            dw       4 dup (?)


        fac_mem         ends

        prog            segment
                        assume   cs:prog,ds:fac_mem ——DS はセグメント fac_mem とする
0000                    push     ds            ——使用するレジスタの PUSH
0001                    push     es
0002                    push     ax
0003                    push     cx
0004                    push     di
0005                    push     si

0006    begin:          mov      ax,fac_mem    ——DS のセット
0009                    mov      ds,ax
000B                    mov      es,ax         ——ES を DS と同じにする
000D                    cld                    ——DF のクリア(ストリング命令でポインタをインクリメントする)
000E                    mov      cx,4          ——転送ワード数
0011                    lea      si,fac        ——FAC のアドレスを SI にロード
0015                    lea      di,fac2       ——FAC2 のアドレスを DI にロード
0019            rep     movsw                  ——メモリ間転送. FAC の値が FAC2 にコピーされる
001B                    pop      si            ——レジスタの復帰
001C                    pop      di
001D                    pop      cx
001E                    pop      ax
001F                    pop      es
0020                    pop      ds

0021                    iret                   ——BASIC へ復帰

0022    add_ent:        push     di            ——ROM ルーチン単精度加算
0023                    mov      di,2bh
0026                    jmp      short int_c4

0028    sub_ent:        push     di            ——ROM ルーチン単精度減算
0029                    mov      di,2ch
```

```
002C                    jmp     short int_c4

002E    mul_ent:        push    di   ———— ROMルーチン単精度乗算
002F                    mov     di,2dh
0032                    jmp     short int_c4
0034    div_ent:        push    di   ———— ROMルーチン単精度除算
0035                    mov     di,2eh
0038                    jmp     short int_c4

003A    int_c4:         int     0c4h ———— ROMルーチンに内部割込みする
003C                    pop     di
003D                    iret    ———————— BASICへ復帰
003E    prog            ends

                        end
```

6-4-6　ROM内演算ルーチンによる配列データの計算

USR関数は，引数を直接FACに代入してくれるので，大変便利な関数ですが，引数を1個しか許さないため，大量のデータの演算には使い辛い面があります．次に紹介するのは，CALL命令を使って2つの1次元配列の同じ添字を持つ要素に対し，四則演算を行うプログラムです．

リスト6-4-3にそのBASIC部分を，リスト6-4-4にマシン語部分を載せました．

BASICプログラムは，配列A(I)，B(I)にデータを代入した後，加減乗除のどれを行うかを決め，各配列データの先頭をセットして演算に対応するマシン語ルーチンをCALLするものです．

マシン語ルーチンは，CALL命令のパラメータのA(0)，B(0)から得られた配列データの先頭アドレスをもとに，それぞれの配列の先頭からデータを順次取り出して，FAC2とFACに書き込み，指定された演算を行うROMルーチンをCALLします．さらに，FACに返された演算結果を配列C(I)に代入して制御をBASICに戻します．

リスト 6-4-3　配列四則演算BASIC部

```
100 ' L6-4-3
110 ' Sample program of INT c4h
120 '
130 CLEAR ,&H1E00:DEF SEG=&H1E00
140 BLOAD "L6-4-4":CLS
150 INPUT "Enter function (1:ADD,2:SUB,3:MUL,4:DIV,5:END)";FUNC%
160 IF FUNC%<=0 OR FUNC%>=5 THEN *EOP
170 'DATA INPUT
180 INPUT "How many data";N.DATA%:NUMDIM=N.DATA%-1
190 DIM A#(NUMDIM),B#(NUMDIM),C#(NUMDIM)
200 PRINT"Input data is treated as double precision
210 '
220 FOR I=0 TO NUMDIM
230      PRINT"Data A#(";I;"),B#(";I;")";:INPUT A#(I),B#(I)
240 NEXT I
250 MADR=0
260 '
```

```
270 IF FUNC%=1 THEN FUNC$="+":GOTO *CAL
280 IF FUNC%=2 THEN FUNC$="-":GOTO *CAL
290 IF FUNC%=3 THEN FUNC$="*":GOTO *CAL
300 IF FUNC%=4 THEN FUNC$="/":GOTO *CAL
310 '
320 *CAL
330 CALL MADR(A#(0),B#(0),C#(0),N.DATA%,FUNC%)
340 CLS
350 FOR I=0 TO NUMDIM
360     PRINT TAB(2);A#(I);TAB(21);FUNC$;TAB(23);B#(I);
370     PRINT TAB(42);"=";TAB(44);C#(I)
380 NEXT I
390 GOTO 110
400 *EOP:PRINT "Function complete":END
```

リスト 6-4-4 配列四則演算マシン語ルーチン

```
; List 6-4-4
; Program of real number calculation
;                          for List 6-4-3
fac_mem         segment  at 60h ──────── FACのあるセグメントの値
                org      1414h
fac_type        dw       ? ─────────────── 変数の型をセットする場所
                org      1416h
fac             dw       4 dup (?) ─────── FAC(4ワード)
                org      1420h
fac2            dw       4 dup (?) ─────── FAC2(4ワード)
fac_mem         ends
;
funcadd         equ      01h  ┐
funcsub         equ      02h  │ 引数 FUNC%の値を定数として設定
funcmul         equ      03h  │
funcdiv         equ      04h  ┘
;
addvct          equ      05h  ┐
subvct          equ      30h  │ 各演算に対応したROMルーチン・ディスパッチャ
mulvct          equ      06h  │ のルーチン番号
divvct          equ      07h  ┘
;
var_type        equ      08h ──────────── 倍精度型実数値のタイプ
;
prog            segment ───────────────── プログラムのセグメント
                assume   cs:prog,ds:prog
main:
                push     ds   ┐
                push     es   │
                push     ax   │
                push     bx   │ レジスタの退避(最低でもセグメント・レジスタと
                push     cx   │ AX,BXはPUSHする)
                push     di   │
                push     si   ┘
                                   ;start main ──── メインルーチンのスタート
                les      si,[bx] ──────── CALLの引数のアドレステーブルの最初のアドレスをES:SIにロード
                mov      ax,es:[si] ───── 最後の引数(FUNC%)の値をAXにロード
                mov      cs:func,ax ───── FUNC%の値を変数funcに退避
                les      si,4[bx] ─────── 引数N.DATA%のアドレスをES:SIにロード
                mov      cx,es:[si] ───── N.DATA%の値をCXにロード(LOOP命令で使用)
```

```
        add     bx,8            ── BXを引数A#(0)のアドレスの入ったテーブル中のアドレスにセット
        push    ds              ┐ 引数A#(0),B#(0),C#(0)のアドレス・テーブルのア
        push    bx              ┘ ドレスをスタックへPUSH
        mov     bp,sp           ── BPはスタック中のアドレス・テーブルを格納したメモリを指す
;
        xor     dx,dx           ── DXはアドレスポインタとして使用するので初期化する(DX=0)
data_set:
        mov     ax,fac_mem      ┐ DS=60H(FACのあるセグメント)に設定
        mov     ds,ax           ┘
        lea     bx,fac_type     ┐ FACに入る数値の型のセット
        mov     word ptr [bx],var_type ┘
;
A#(1)をFAC2へ転送する
        lds     bx,[bp]         ── スタック中のアドレス・テーブルのアドレスをDS:BXにロード
        lds     si,8[bx]        ── DS:SIにA#(0)のアドレスをセット      ┐
        add     si,dx           ── 演算すべき配列データのアドレスを求める │
        mov     ax,fac_mem      ┐ ESに60Hをセット                     │ MOVS命令の準備
        mov     es,ax           ┘                                     │
        lea     di,fac2         ── DIにFAC2のアドレスをセット          ┘
        call    trans8b         ── 配列A#(1)の値をFAC2へ転送
;
B#(1)をFACへ転送する
        lds     bx,[bp]         ── アドレステーブルのアドレスをロード
        lds     si,4[bx]        ── DS:SIにB#(0)のアドレスをセット
        add     si,dx           ── 演算すべき配列データのアドレスを求める
        mov     ax,fac_mem      ┐ ESに60Hをセット
        mov     es,ax           ┘
        lea     di,fac          ── DIにFACのアドレスをセット
        call    trans8b         ── 配列B#(1)の値をFACへ転送する
;
        call    calc            ── 演算サブルーチン(ROMルーチンのコール)
;
演算結果をFACからC#(1)に転送する
        mov     ax,fac_mem      ┐ DSに60Hをセット
        mov     ds,ax           ┘
        lea     si,fac          ── SIにFACのアドレスをセット
        les     bx,[bp]         ── 引数のアドレステーブルのアドレスをES:BXにロード
        les     di,es:[bx]      ── 配列C#(0)のアドレスをセット
        add     di,dx           ── 結果の格納される配列データのアドレスを求める
        call    trans8b         ── FACから結果をC#(1)へ転送
;
        add     dx,8            ── 配列データのアドレス・ポインタの更新
        loop    data_set        ── CX(N.DATA%)が0になるまで繰り返す
;
        pop     bx              ┐ BX,DSの復帰
        pop     ds              ┘
                        ;end of main
        pop     si              ┐
        pop     di              │
        pop     cx              │
        pop     bx              │ レジスタの復帰
        pop     ax              │
        pop     es              │
        pop     ds              ┘
        iret                    ── BASICへの復帰
;
trans8b:
        push    cx
        cld                     ── 転送方向設定
        mov     cx,4            ── 転送ワード数
rep     movsw                   ── 転送
        pop     cx
        ret
```

```
;
calc:
                push    ds          ROMルーチンを使用する際は破壊するとまずいレジスタ
                push    di          はすべて PUSH しておくこと（どのレジスタが使用さ
                push    ax          れるか不明なので，すべてのレジスタをPUSHする方が
                push    bx          無難）
                push    cx
                push    dx
                mov     ax,fac_mem  DS=60Hにセット．ROMルーチンコール時には，DS
                mov     ds,ax       はテキスト・セグメントにおく
                mov     ax,cs:func
                mov     di,addvct
                cmp     ax,funcadd
                je      int_go
                mov     di,subvct
                cmp     ax,funcsub  DIにROMルーチン・ディスパッチャのルーチン番号をセ
                je      int_go      ット後FUNC%の値をチェックして同じならint_goへ
                mov     di,mulvct
                cmp     ax,funcmul
                je      int_go
                mov     di,divvct
                cmp     ax,funcdiv
                je      int_go
                jmp     out_calc ── FUNC%が1～4でないときは何もせず終る
;
int_go:
                int     0c4h ────── 内部割込み
out_calc:
                pop     dx
                pop     cx
                pop     bx
                pop     ax          レジスタのPOP
                pop     di
                pop     ds
                ret
;
func            dw      0 ───────── FUNC%の退避場所
;
prog            ends
                end
```

7章
ハードウェア構成と割込み

マシン語プログラムは，ビット操作の容易さやスピードが速いという特徴を活かし，ハードウェアを直接アクセスすることが可能になります．しかし実際にプログラムを作るには，そのハードウェアに関する知識がある程度必要です．この知識は，ハードウェアを作りあげるためのものではなく，ハードウェアをプログラムで操作する際に必要なものです．そのためには，各ハードウェアのマニュアルを読むのが一番ですが，その基礎説明ということで，この章では，PC-9801のハードウェアと，ハードウェア割込みという方法について概説します．

7-1　PC-9801のハードウェア様式

7-1-1　ハードウェア構成

PC-9801シリーズにはいくつかのバリエーションがありますが，基本的なハードウェア構成はみな同じです．図7-1-1に本体のハードウェア・ブロック図を示します．この図の各ブロックには，その機能とそれを実現しているLSIの型番が書かれており，データの大まかな流れが矢印で表されています．LSI自体はハード(金物)として固定されたものですが，その動作はソフトウェアで設定できるものがほとんどです．

図中，太い矢印は"バス"と呼ばれるアドレスやデータ，制御などの信号線の集まりです．CPUと各LSIは，バスを介してパラレルに必要な信号線が接続されています．CPUは通常，このバスの主導権を握っており，データは，一度CPUのレジスタに読み込まれてから，各LSIへ送られます．たとえば，メモリ中のデータをプリンタへ出力する場合，メモリから直接プリンタI/FのμPD8255へデータが流れるのではなく，メモリ→CPU→プリンタI/Fというルートをたどります．

ただしDMAという手法が行われる時はこの限りではありません．

各ブロックのデバイスを使う場合には，それぞれのデバイスの使い方と，それが9801でどのように使われているかを知る必要があります．

我々が今まで扱ってきたのは，このうち主にCPUとメモリでした．コンピュータを形づくっている大きな要素で残っているのは，入出力に関する部分です．これには，キーボード，ディスプ

レイ，RS-232C，プリンタ，GPIB，フロッピー・ディスク，などが考えられますが，それぞれの機能の実現のために専用のLSIが使われているため，入出力のハードウェアを扱うにはこれらのLSIの使い方を知らなければなりません．

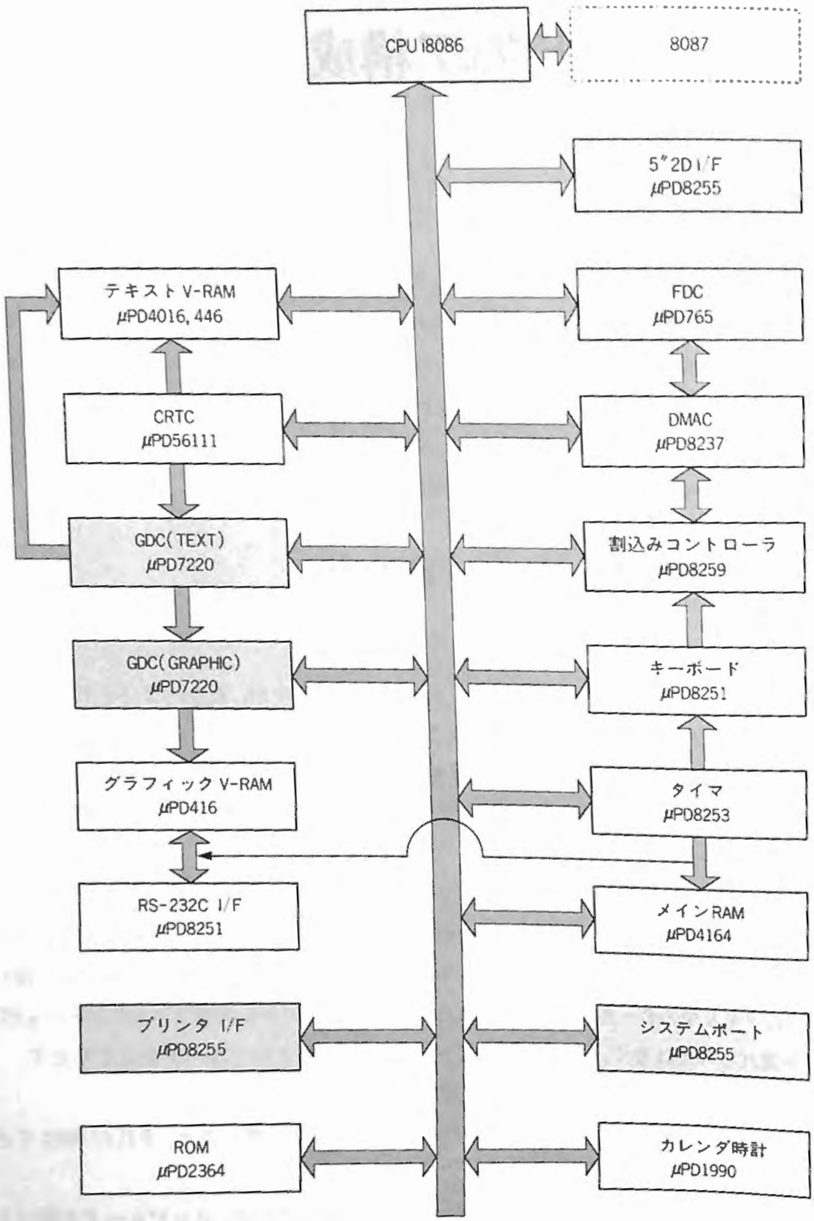


図7-1-1　ハードウェアブロック図

7-1-2　入出力に使用される主なデバイス

図7-1-1のブロックのうち，入出力に使用されるLSIについて簡単に説明します．

○μPD8255

8255は通常パラレルポートとして用いられます．パラレル方式とは，一度に複数のビットのデータをやり取りする方法で，ビット列を時間軸に展開して1ビットずつ送るシリアル方式と区別されます．8255には，図7-1-2に示すように8ビットのポートがA，B，Cの3つ用意されています．

PC-9801では，3個の8255をプリンタ・インターフェイス，5インチ2Dディスク・インターフェイス，システムポート（BEEPの設定など）に使用しています．

○μPD8251A

8251は，前述のシリアル伝送用のポートとして使用されます．これは，USART（ユニバーサル・シンクロナス・アシンクロナス・レシーバ・トランスミッタ）とも呼ばれ，現在使用されている多くのシリアルデータ伝送を行うことができます．

PC-9801では，キーボード・インターフェイスと，RS-232Cインターフェイスにそれぞれ1個ずつ使用されています．

○μPD7220（GDC)

μPD7220は，GDC（グラフィック・ディスプレイ・コントローラ）とも呼ばれ，ディスプレイ・コントロール用にNECが開発した高機能CRTC(カソード・レイ・チューブ・コントローラ)です．

従来のCRTCと異なり，直線，円弧などのグラフィック・パターンを，命令によって高速に描画させることができます．

PC-9801では，このGDCを，キャラクタ用と，グラフィック用に2個搭載しています．

○μPD765

μPD765は，FDC（フロッピーディスク・コントローラ）とも呼ばれ，その名の通りフロッピー・ディスクのコントロール用に開発されたチップです．

PC-9801では，5インチ2D以外のすべてのフロッピーディスクのインターフェイスに使用されています．また，5インチ2D用のディスク・ドライブである，PC-80S31にも，このチップが使用されています．

○μPD8259A

μPD8259Aは，後述のCPUに対する外部からの割込みの優先順位などをコントロールするためのチップで，割込みコントローラとも呼ばれます．

1個の8259Aで8レベル(8段階)の割込みレベルを設定することができ，PC-9801ではこのチップを2個使用することで，15レベルの割込み順位を設定しています．

7-1-3 I/Oポートと入出力命令

PC-9801では，これまで述べた各デバイスチップとの入出力に関するデータのやりとりを，I/Oアドレス空間に設定したI/Oポートを介して行います．図7-1-2に9801のポートアドレス表を示します．

	ポートアドレス								装置名
	7	6	5	4	3	2	1	0	
1	0	0	0	0	0	×	A_0	0	割込みコントローラ μPD8259A(マスタ)
2	0	0	0	0	1	×	A_0	0	割込みコントローラ μPD8259A(スレーブ)
3	0	0	0	A_3	A_2	A_1	A_0	1	DMAコントローラ μPD8237A-5
4	0	0	1	0	×	×	×	0	カレンダ時計 μPD1990
5	0	0	1	0	×	A_1	A_0	1	DMAバンク
6	0	0	1	1	×	×	A_0	0	RS-232Cインターフェイス μPD8251A
7	0	0	1	1	×	A_1	A_0	1	システム・ポート μPD8255A-5
8	0	1	0	0	×	A_1	A_0	0	プリンタ・インターフェイス(セントロ) μPD8255A-5
9	0	1	0	0	×	×	A_0	1	キーボード・インターフェイス μPD8251A
10	0	1	0	1	×	×	A_0	0	NMIコントロール
11	0	1	0	1	×	A_1	A_0	1	5インチフロッピーディスクI/F μPD8255A-5
12	0	1	1	0	A_2	A_1	A_0	0	CRTコントローラ μPD7220(テキスト)
13	0	1	1	1	A_2	A_1	A_0	0	CRTコントローラ μPD52611
14	0	1	1	1	×	A_1	A_0	1	タイマコントローラ μPD8253-5
15	1	0	0	0	0	0	A_0	0	5インチ固定ディスクI/F
16	1	0	0	1	×	A_1	A_0	0	8インチフロッピーディスク・コントローラ μPD765A
17	1	0	0	1	0	A_1	A_0	1	カセットMTインターフェイス μPD8251A
18	1	0	0	1	1	0	A_0	1	GPIBスイッチ
19	1	0	1	0	A_2	A_1	A_0	0	CRTコントローラ μPD7220(グラフィック)
20	1	0	1	0	A_2	A_1	A_0	1	文字パターンROM
21	1	1	0	0	0	A_1	A_0	0	プリンタ・インターフェイス(ODA) μPD8255A-5
22	1	1	0	0	1	A_1	A_0	0	5インチ2DDフロッピーディスク・コントローラ μPD765A
23	1	1	0	0	A_2	A_1	A_0	1	GPIB μPD7210
24	1	1	1	0	0	0	0	0	キーボード(スキャン方式)
				～					
	1	1	1	0	1	0	1	1	

※$(D0)_{16}$～$(DF)_{16}$，$(EC)_{16}$～$(F0)_{16}$はユーザーが自由に使用できるI/Oポートアドレスである．これらを除いたすべてがシステムで使用済みか予約(Reserved)されている

図7-1-2　ポートアドレス表

図中，たとえばセントロニクス・プリンタ・インターフェイスのポートアドレスは，

$0100 \times A_1 A_0 0$

と示されています．このうちA_1, A_0というビットはそのインターフェイスLSIであるμPD8255Aの内部のアドレスを指定するものです．また×は，0でも1でも良いことを示していますから，本書では0として指定することにしますと，この8255Aに割り当てられたポートアドレスは，

01000000＝40H
01000010＝42H
01000100＝44H
01000110＝46H

の4つであることがわかります.

マシン語でこれらのI/Oポートをアクセスするには, 入出力命令を使います. 入出力命令には, INとOUTの2つがあり, それぞれ入力 (ポートを読む), 出力 (ポートに書く) に対応しています.

IN命令のバリエーションを図7-1-3(a)に示します. ディスティネーションすなわちデータが取り込まれる対象は, ALまたはAXのみに限られています. ポートアドレスの指定には, そのアドレスを直接与える方法と, DXにあらかじめアドレスをセットしておいて間接にそれを与える方法の2通りがありますが, 前者の場合には, 0~255までのアドレスしか指定できません.

出力用のOUT命令はIN命令のオペランドの順序を入れ換えて考えてください(図7-1-3(b)).

	命　令	意　味
(a)入力命令	IN AX, D0H	ポート・アドレスD0Hからワードデータを AXに入れる (ポート・アドレスD0Hの内容をALに, D1Hの内容をAHに入れる)
	IN AL, 44H	ポート・アドレス44Hからバイトデータを ALに入れる
	IN AX, DX	DXの内容が示すポートアドレスからワードデータを AXに入れる
	IN AL, DX	DXの内容が示すポートアドレスからバイトデータを ALに入れる
(b)出力命令	OUT D0H, AX	AXの内容(ワードデータ)をポート・アドレスD0H, D1Hに送る (AL→D0H, AH→D1H)
	OUT 46H, AL	ALの内容(バイトデータ)をポート・アドレス46Hに送る
	OUT DX, AX	AXの内容(ワードデータ)をDXの内容が示すポート・アドレスに送る
	OUT DX, AL	ALの内容(バイトデータ)をDXの内容が示すポート・アドレスに送る

図7-1-3　入出力命令の例

入出力命令を使った例として, PC-9801のBEEP音を直接出すプログラムを紹介します. BEEP音は, I/Oポート・アドレスの37Hに06HをOUTすればON, 07HをOUTすればOFFになります (リスト7-1-1).

リスト 7-1-1　OUT命令の使用例

```
; List 7-1-1
; Progragm of beep sound

num_lop equ     20  ――― 音の繰り返しの回数
num_bel equ     20  ――― 音のON,OFFの回数
beepon  equ     06h ――― Beep音ONのデータ
beepoff equ     07h ――― Beep音OFFのデータ
;
main    segment
        assume  cs:main
start:
        push    ax    ┐
        push    bx    │レジスタ退避
        push    cx    ┘
        xor     bx,bx ――― BXのクリア
```

```
        mov     bh,num_lop
begin1:
        mov     bl,num_bel
begin2:
        mov     cx,0fffh ──┐── ループの回数(待ち時間)
        mov     al,beepon  │
        out     37h,al     │ 約4000回分Beep音ON
wait1:                     │
        loop    wait1     ─┘
;
        mov     cx,0fffh  ─┐
        mov     al,beepoff │
        out     37h,al     │ 約4000回分Beep音OFF
wait2:                     │
        loop    wait2     ─┘
;
        dec     bl ──────── BL=0になるまで20回のON,OFF
        jne     begin2
        mov     cx,0ffffh ─┐
wait3:                     │ 65535回の間Beep音OFF
        loop    wait3     ─┘
        dec     bh ──────── BH=0になるまで音の点滅を20回繰り返す
        jne     begin1
;
        pop     cx        ─┐
        pop     bx         │ レジスタ復帰
        pop     ax        ─┘
        iret ────────────── BASICへ復帰
;
main    ends
        end
```

7-2 ハードウェア割込み

7-2-1 割込みとは

コンピュータの"割込み"とは，何らかの要因によって，現在実行中のルーチンをいったん中断して他のルーチン（割込み処理ルーチン）を実行し，それが終わるとまた元のルーチンに戻るという処理を指します（図7-2-1）．

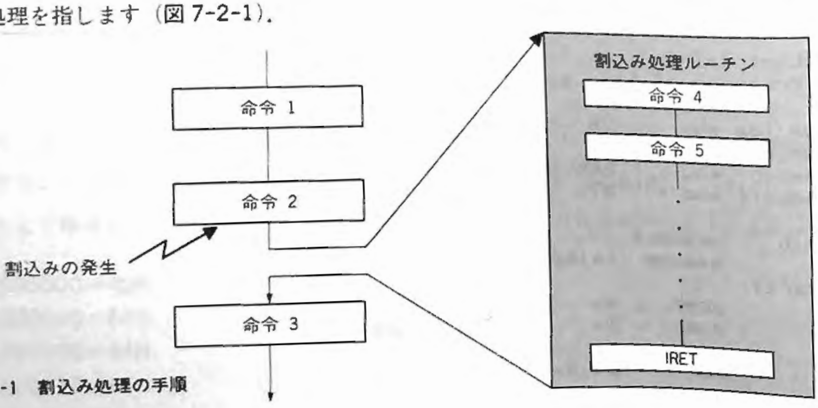


図7-2-1 割込み処理の手順

8086には，第4章で述べたINT命令による内部割込みと，これから述べる外部割込みという2つの割込み方法があります．前者はサブルーチン・コールの変形に近いものでしたが，後者は，CPUの外，すなわち外部からの要因によって引きおこされるもので，ハードウェア割込みとも呼ばれます．

図7-2-2(a)で，もしスイッチがON状態であれば，ある処理をすぐに行わなければならないとすると，CPUはたえずこのI/Oポートを読みだして，ON/OFFを調べるという図7-2-2(b)のようなプログラムを実行していなければなりません．この場合，スイッチがON/OFFされるタイミングがCPUの実行スピードに比べて格段に遅い場合，(b)図のループ回数は大きくなり，無駄が生じます．

しかし図7-2-3のようにONの状態になったことを示す信号線をCPUの割込み入力端子に接続して，外部割込みの機能を使うようにすると，図7-2-1のようなプログラムにすることができ，

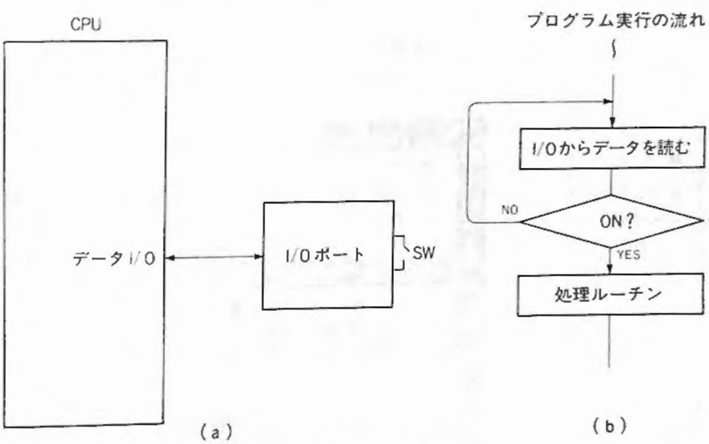


図7-2-2　ループ・プログラムによるキー入力

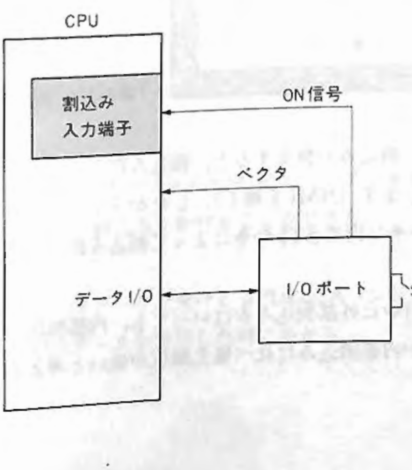


図7-2-3　外部割込みによる入力

CPUを効率よく使うことが可能になります．外部割込みではまた，外部からの信号に対してすばやく反応できるため，より高速な入出力を行うことができます．

割込み処理ルーチンの先頭アドレスは，内部割込みの場合と同じ領域に書かれた値（ベクタ）で参照されます．ただし，この値（ベクタ）は割込みの要因によって外部からハードウェア的にCPUに与えられるところが，INT命令のオペランドをベクタとした内部割込みと異なる点です．

外部割込みの中には，CPUのIF（インタラプト・フラグ）によってマスク（無視）できるものと，そうでないものがあります．後者はNMI（ノン・マスカブル・インタラプト）と呼ばれ，ベクタが決められているため，外部からベクタを与える必要がありません．

このNMIはメモリのチェックなどの特別な場合に用いられるので，ユーザーの自由にならないため本書では扱わないことにします．

前者のマスク可能な割込みによるCPUの動きを図7-2-4に示します．

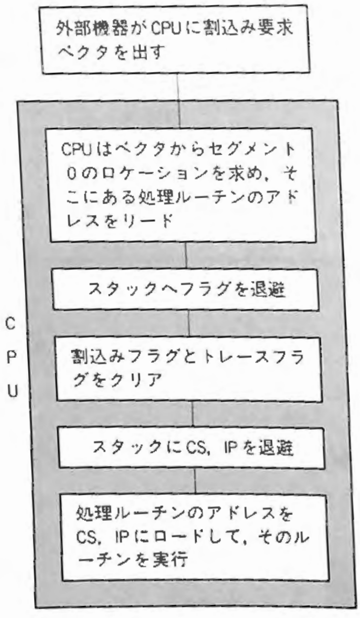


図7-2-4　割込みのシーケンス

ところで，内，外部の両割込みともに，割込みが発生すると，割込み許可フラグをクリアし，以後の外部割込みを受け付けないようにします（NMIを除く）．したがって，外部割込みによる多重割込みをする場合は，割込み処理ルーチン内でSTI命令によって割込み許可フラグをセットしてやる必要があります．

また，内部割込みを発生させる命令実行中に外部割込みがはいっても，内部割込みが優先されます．このことから，外部割込みのほうが内部割込みに比べ優先順位が低いと考えることができます．

7-2-2　PC-9801の割込み

PC-9801では，外部割込みのベクタを発生させるために，前述の8259Aという割込みコントローラを2個使用しています．8259Aには，8本の割込み入力端子があり，8個までの割込み要因のベクタの発生と，優先順位の設定,マスクなどを行うことができます．PC-9801の2個の8259Aは，マスタとスレーブという従属関係で，マスタの割込み入力端子の1本がスレーブの出力端子と接続された，カスケード接続になっており，図7-2-5に示すような割込みを受けつけるようになっています．

デバイス名	割込み要求信号	レベル*	割込み名	データビット 7	6	5	4	3	2	1	0	ベクタ番号
μPD8259A (マスタ)	IR0	0	タイマ	T_7 0	T_6 0	T_5 0	T_4 0	T_3 1	0	0	0	08
	IR1	1	キーボード	0	0	0	0	1	0	0	1	09
	IR2	2	CRTV	0	0	0	0	1	0	1	0	0A
	IR3	3	拡張スロット　INT0	0	0	0	0	1	0	1	1	0B
	IR4	4	RS-232C	0	0	0	0	1	1	0	0	0C
	IR5	5	拡張スロット　INT1(CMT)	0	0	0	0	1	1	0	1	0D
	IR6	6	拡張スロット　INT2(PRODA)	0	0	0	0	1	1	1	0	0E
	IR7	7	スレーブ	0	0	0	0	1	1	1	1	0F
μPD8259A (スレーブ)	IR8	8	セントロプリンタ	T_7 0	T_6 0	T_5 0	T_4 1	T_3 0	0	0	0	10
	IR9	9	拡張スロット　INT3(5"HD)	0	0	0	1	0	0	0	1	11
	IR10	10	拡張スロット ♯1, ♯2, ♯3 INT41(5"FD-2DD)	0	0	0	1	0	0	1	0	12
	IR11	11	拡張スロット ♯4 INT42(8"FD)	0	0	0	1	0	0	1	1	13
	IR12	12	拡張スロット　INT5	0	0	0	1	0	1	0	0	14
	IR13	13	拡張スロット　INT6	0	0	0	1	0	1	0	1	15
	IR14	14	8087	0	0	0	1	0	1	1	0	16

＊割込みレベルは変更可能

図7-2-5　PC-9801の割込み要因

図7-2-5の割込み名はそれぞれ次のような要因による割込みの名称です．

○タイマ

μPD8253C(プログラマブルタイマ)のカウンタ0からの割込み．8253Cにカウント値を設定して，インターバルな割込みをかけることができる．

○キーボード

キーボードからのデータを受けるμPD8251A（シリアル・インターフェイス）からの割込み．キーボードからのデータを受信した時にかかる．

○ CRTV

ディスプレイ・タイミングの垂直帰線区間にはいったことを示すCRTCからの割込み．1秒に約60回かかる．一度発生するとI/Oの64Hに書込みを行わないと以後発生しない．帰線区間では，CRTディスプレイの見えない部分を電子ビームが走査しているので，この期間にV-RAMへのアクセスを行えば，画面のちらつきを生じない．

○ RS-232C

μPD8251AのTXRDY，TXEMPTY，RXRDYによる割込み．システム・ポート（8255A）のPC2，1，0により制御可能．

○セントロニクス・プリンタ

プリンタポート8255Aのポート3からの割込み．

○ディスク装置

5インチHD，5インチ2DD，8インチ2D等のディスク・ドライブからの割込み．

○ 8087

8087（数値演算プロセッサ）の演算で例外の検出，処理に使われる．

8259Aには，図7-2-6に示すように，それぞれ2つのI/Oアドレスが割り当てられていて，マシン語のIN，OUT命令による読み書き（Read，Write）でデータをやり取りします．この図で，同じアドレスに対する読み書きなのにそのコマンドの種類が異なっているものがありますが，これは間違いではありません．

これらはデータの内容によって区別されたり，ある決まったシーケンスによるアクセスによってどのコマンドであるかが認識されるのです．

8259Aは，システム起動時にROM内のルーチンによって初期化され，機能が設定されます．初期化は，図7-2-6のICW（イニシャライズ・コマンド・ワード）のデータを8259Aに送ることにより行われます．

初期化の終わった8259Aに対しては，OCW（オペレーション・コマンド・ワード）データの書込みや各リード・コマンドをいつでも利用することができますが，本書では紙面の関係でマスターの割込みマスクと，EOIという処理しか使っていません．他にも優先順位の変更やポーリング・モードなどいろいろな機能が備わっていますので，割込み処理を十分に活かしたい場合にはμPD8259Aのマニュアルを一度じっくり調べてください．

割込みマスクは，OCW1コマンドとIMRリードコマンドによって行います．このコマンドで，やり取りされるデータのビットがセットされているときは，その要求がマスクされることを示します．通常は，IMRリード・コマンドで現在のマスク状態を読み出し，必要なビットのセット，リセットをOCW1コマンドで書き込みます．

8259Aに対して同時に複数の割込み要求があると，IRR（割込み要求レジスタ）の内容から最も優先順位の高い割込みが決定され，8259Aはそのベクタをバスに出力します．またその際ISR（イ

ン・サービスレジスタ）がセットされて，以後は，それと同じか，または低い順位の割込みは受け付けられなくなります．そこで，その割込み処理ルーチンを終了する前や，必要に応じてEOI（割込み終了）コマンドを出してISRのリセットをし，他の割込みに対応できるようにします．

デバイス名	命　令	I/Oポート・アドレス	R/W	データビット 7	6	5	4	3	2	1	0	備　考
マスタ	ICW1	00	W	0	0	0	1	LTIM	0	S	1	S＝0
	ICW2	02	W	T_7	T_6	T_5	T_4	T_3	0	0	0	T_7～T_3 ＝00001
	ICW3	02	W	1	0	0	0	0	0	0	0	
	ICW4	02	W	0	0	0	SFNM	BUF	1	0	1	BUF＝1
	OCW1	02	W	M_7	M_6	M_5	M_4	M_3	M_2	M_1	M_0	
	OCW2	00	W	R	SL	EOI	0	0	L_2	L_1	L_0	
	OCW3	00	W	0	ESMM	SMM	0	1	P	RR	RIS	
	ポールモード	00	R	I	X	X	X	X	W_2	W_1	W_0	
	IRRリード	00	R	IR7	IR6	IR5	IR4	IR3	IR2	IR1	IR0	
	ISRリード	00	R	IS7	IS6	IS5	IS4	IS3	IS2	IS1	IS0	
	IMRリード	02	R	M_7	M_6	M_5	M_4	M_3	M_2	M_1	M_0	
スレーブ	ICW1	08	W	0	0	0	1	LTIM	0	S	1	S＝0
	ICW2	0A	W	T_7	T_6	T_5	T_4	T_3	0	0	0	T_7～T_3 ＝00010
	ICW3	0A	W	0	0	0	0	0	1	1	1	
	ICW4	0A	W	0	0	0	SFNM	BUF	0	0	1	BUF＝1
	OCW1	0A	W	M_{15}	M_{14}	M_{13}	M_{12}	M_{11}	M_{10}	M_9	M_8	
	OCW2	08	W	R	SL	EOI	0	0	L_2	L_1	L_0	
	OCW3	08	W	0	ESMM	SMM	0	1	P	RR	RIS	
	ポールモード	08	R	I	X	X	X	X	W_2	W_1	W_0	
	IRRリード	08	R	IR15	IR14	IR13	IR12	IR11	IR10	IR9	IR8	
	ISRリード	08	R	IS15	IS14	IS13	IS12	IS11	IS10	IS9	IS8	
	IMRリード	0A	R	M_{15}	M_{14}	M_{13}	M_{12}	M_{11}	M_{10}	M_9	M_8	

図7-2-6　μPD8259Aのポート・アドレスとコマンド

図7-2-7にPC-9801での初期化の設定を参考までに示しておきます．

① μPD8259A マスタの初期化例	② μPD8259A スレーブの初期化例
MOV AL, 11H	MOV AL, 11H
OUT 00H, AL	OUT 08H, AL
MOV AL, 08H	MOV AL, 10H
OUT 02H, AL	OUT 0AH, AL
MOV AL, 60H	MOV AL, 07H
OUT 02H, AL	OUT 0AH, AL
MOV AL, 1DH	MOV AL, 09H
OUT 02H, AL	OUT 0AH, AL

図7-2-7 μPD8259Aの初期設定

7-2-3 割込みを使ったプログラムの実際

ハードウェア割込みの処理は，割込みの要因となった入出力機器などに対して行うものと，要因となった機器以外に対して行うものがあります．たとえば，キーボードが押されたことによって発生した割込みで，そのキーデータを取り込むという処理を行う場合は，要因と処理の対象が同じですが，タイマなどによる，一定周期の割込みによって，キーボードからデータを取り込む場合などは，要因と対象が別です．

要因と対象が同じ割込みとして，キーボードからの割込みの簡単な例をひとつ紹介しましょう（リスト7-2-1）．

リスト 7-2-1 割込みの使用例

```
0000  FB            sti
0001  F4            hlt
0002  F4            hlt
0003  F4            hlt
0004  F4            hlt
0005  F4            hlt
0006  F4            hlt
0007  F4            hlt
0008  CF            iret
```

このプログラムは見ての通り，1番地からHLT命令が入っています．一般に，HLT命令が実行されると，CPUはホールト（実行停止）状態になります．

しかしこのプログラムでは，あらかじめ，STI (Set-Interrupt) 命令によりCPUの割込みフラグを許可してあるため，プログラムの実行後，キーボード等からの入力（割込み）があるとホールト状態をぬけ，それぞれの割込み処理をしたのち，HLTの次の命令を実行することができます．

PC-9801でキーボードからの割込みが生じたときは，割込み処理として入力データの取込みを行いますが，表面上は見ることができません．デバッガ（または機械語モニタ）でこのプログラ

ムを入力したのち，

　　－G=0，8⏎　（h] G0，8⏎）

として実行してください．一時的にホールト状態になりますが，キーボードを数回たたくとデバッガ（モニタ・モード）に戻ります．

ここで，レジスタの内容を見てみると，割込み許可フラグがセットされているのがわかります（リスト 7-2-8）．

```
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0DEC  ES=0DEC  SS=0DEC  CS=0DEC  IP=0008   NV UP EI PL NZ NA PO NC
0DEC:0008 CF              IRET
                                              割込み許可フラグがセットされている
```

図7-2-8　フラグの状態

PC-9801のキーボードは，キーを押した時だけでなく，離した時もデータを出力する（割込みがかかる）ので，一度のキー入力で2つのHLTが解除されます．

なお，割込みは，キーボードだけでなく，ディスクや，プリンタにも許可されているため，ディスク・ドライブ（8インチ，5インチ2HD）の開閉によっても，ホールトからぬけることができます．

7-2-4　多重処理プログラム

一方，一定周期の割込みを使う例には，マルチタスクなどの多重処理があります．これは，入出力に限らず，すべての処理をある単位時間で切り換えて，見かけ上，同時に2つ以上の処理を行うようにするもので，TSS（タイム・シェアリング・システム）と呼ばれる処理にも用いられています．

ここでは，割込みを使ったプログラムの例として，この一定周期の割込みによるプリンタ出力プログラムを紹介します．

パソコンで使用されるプリンタ（PC-PR201等）の印字速度は，1秒間に100文字前後というのが普通ですが，これはCPUの処理速度からみれば，非常に遅いスピードです．このためプリンタの印字中，CPUはこの処理に時間を取られ，他の処理がほとんどできなくなります．そこで，プリンタへの出力をバックグラウンドで行うようにして，表向きは他の処理ができるようにしたのがリスト 7-2-2のマシン語ルーチンです．リスト 7-2-3は，このマシン語ルーチンの起動用プログラムです．

これは，BASIC部分のプログラムにより，アスキー・セーブされたBASICプログラム・ファイルを読み出してV-RAMへ転送しておき，その後，割込みのたびにプリンタへ1文字ずつ出力するというものです．BASICのコマンドレベルでサポートしていないため，あまり実用にはなりませんが，多重処理の例としてわかりやすいと思います．

一定周期の割込みには，タイマは使わずにCRTVの割込みを使っています．これは，ディスプレイ・タイミングの垂直帰線区間にはいったことを示すもので，60分の1秒に1回の割合で発生します．これでは，1秒間に最大60文字までしか出力できず，プリンタの印字速度よりも遅くなるわけですが，実際には，ライン・フィード やキャリッジ・リターンなどの動作に時間がかかるため，これぐらいの周期で十分です．

プリンタにデータを出力するには，BIOSのルーチンをコールする方法と直接ポートを操作する方法がありますが，ここでは後者の方法で行っています．プリンタ・ポートの図7-2-9に示す信号を使って，図7-2-10のように，BUSYをチェックしてデータを出力し，PSTBを操作するという方法によりプリンタにデータを送出します．

命令	I/Oポート・アドレス	R/W	データビット 7	6	5	4	3	2	1	0	備考
モード セット	46	W	1	0	0	0	0	0	1	0	8255Aのモードセット
ライト ストローブ	46	W	0	0	0	0	1	1	1	D	$\overline{PSTB}$のON/OFF D=0 OFF,D=1 ON
	46	W	0	0	0	0	0	1	1	D	IR8のON/OFF D=0 OFF,D=1 ON
ライト インタラプト	44	W	$\overline{PSTB}$	0	0	0	IR8	0	0	0	$\overline{PSTB}$, IR8は本命令でもON/OFF可
ストローブ／インタラプト	40	W	WD8	WD7	WD6	WD5	WD4	WD3	WD2	WD1	プリンタにデータを送る
リード データ（診断用）	40	R	WD8	WD7	WD6	WD5	WD4	WD3	WD2	WD1	コード40で8255AにセットしたWD8～1を引き取る
リード ビジィ	42	R	TYP1	TYP0	MOD	×	×	$\overline{BSY}$	×	×	プリンタの状態を取り込む モードおよびタイプを取り込む

×印） 不定
IR8 ） プリンタ制御回路から8259への割込み信号である

＊PC-9801Fにのみ適用

図7-2-9 プリンタポートの概要

このプログラムでは，割込みが発生すると，BUSYをチェックし，BUSYであればそのまま何もせずに戻り，そうでなければ，バッファ（V-RAM）のデータを1バイト送ります．

V-RAMはA8000Hをベースとするセグメントを使って，オフセット0，1H番地にバッファにためられたデータの大きさ，2，3H番地に次の送出データのアドレスを保持し，それ以降をデー

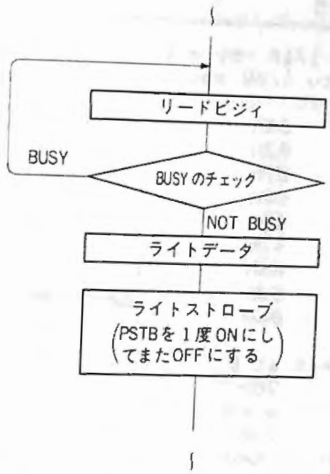


図7-2-10　通常のプリンタ出力シーケンス

タ領域として使用しています.

プログラムは大きく，割込み機能を使うための前処理ルーチン，割込み処理ルーチン，後処理ルーチンに分けることができます.前処理では,ベクタの変更とCRTV割込みのマスク解除,CRTVを発生するようにするための処理を行っています.通常CRTV割込みはマスクされていて，発生しないようになっているため，ベクタ変更部分の割込みフラグのマスクは必要ないわけですが，このような処理の際の一般的な例としてマスクを行います.

マスクの解除は8259AのIMRを読み,2ビット目をAND命令でリセットして出力することにより行っています.CRTRESへの出力命令は，CRTV割込みを発生させるために行うもので，出力データの内容は関係ありません.

前処理はIRETでBASICへ戻って終わります.

割込み処理ルーチンは，CRTV割込みがかかるたびに実行される部分です.このルーチンの先頭アドレスは,前処理ルーチンでベクタとして設定されています.このルーチンは主にプリンタへの出力に関するものです.最後に8259Aに対してEOIという割込み処理ルーチンの終了を示すコマンドとGDCに次の割込みをおこすための出力命令を送り,割込み許可フラグをセットして戻ります.

処理ルーチンで，プリンタに送るデータがなくなった場合，後処理ルーチンが実行され，前処理で変更した部分が元に戻されます.

リスト 7-2-2 プリンタスプーラ

```
;        spool program 1/14  by m.f.
;        interrupt: crtv 1/60 sec
;
crtres           equ     64h        GDCのインタラプトリセットを行うためのポート・アドレス
mimr             equ     02h        マスター8259Aのポート・アドレス
mocw1            equ     02h
mocw2            equ     00h
pdata            equ     40h        プリンタのポート・アドレス
pbusy            equ     42h
pstrb            equ     46h
cr               equ     0dh        改行(CR)，復改(LF)のデータ
lf               equ     0ah
;
system_seg       segment at 0       CRTV割込みのベクタが格納されているアドレス
                 org     28h
crtv_adr         label   word
                 org     2ah
crtv_seg         label   word
system_seg       ends
;
vram             segment at 0a800h  A8000Hに出力データの最終アドレス，A8002Hに次に出力するデータのアドレスをそれぞれ保持
                 org     0
end_point        dw      ?
pointer          dw      ?
st_point         label   byte
vram             ends
;
main_seg         equ     1b00h
;
main    segment
        assume  cs:main,ds:system_seg
start:  push    ds
        mov     ax,system_seg       CRTVベクタの書き換えと保存
        mov     ds,ax
        cli
        mov     ax,main_seg
        xchg    ax,crtv_seg
        mov     cs:save_seg,ax
        mov     ax,offset entry
        xchg    ax,crtv_adr
        mov     cs:save_adr,ax
        mov     al,0                直前に出力したデータを保存するold_chrをイニシャライズ
        mov     cs:old_chr,al
;
        assume  ds:vram
        mov     ax,vram             pointerをイニシャライズ
        mov     ds,ax
        mov     ax,offset st_point
        mov     pointer,ax
;
        in      al,mimr             CRTV割込みのマスクを解除する
        and     al,0fbh
        out     mocw1,al
        out     crtres,al
        pop     ds
        sti                         割込みを許可してBASICへ戻る
        iret
;
```

```
save_seg        dw      ?       ]CRTV 割込みのベクタを保存する領域
save_adr        dw      ?       ]
old_chr         db      ? ———— 直前に出力したデータが保存される領域
;
entry: ———————————————————————— 割込みがかかるとここにエントリする
        assume  ds:vram
        push    ds
        push    ax
        push    bx
        mov     ax,vram
        mov     ds,ax
        mov     bx,pointer
        in      al,pbusy        ]
        and     al,04h          ]プリンタが busy なら busy1 へ
        je      busy1           ]
        mov     al,cs:old_chr   ]
        cmp     al,cr           ]直前の出力データが CR なら,LF を
        mov     al,lf           ]出力するようにする
        je      chr             ]
        mov     al,[bx]         ]出力データの取り出しとポインタ
        inc     bx              ]のインクリメント
chr:
        mov     cs:old_chr,al   ]
        out     pdata,al        ]
        mov     al,0eh          ]プリンタへデータを出力し,スト
        out     pstrb,al        ]ローブ・パルスを出す
        mov     al,0fh          ]
        out     pstrb,al        ]
        mov     pointer,bx      ]
        mov     ax,end_point    ]ポインタを比較し,出力するデー
        cmp     ax,bx           ]タがなければ stop_spool へ
        je      stop_spool      ]
busy1:  out     crtres,al ———— 次の CRTV 割込みを発生するよ
                                うに GDC へコマンドを送る
busy:   mov     al,20h          ]8259A に EOI コマンドを出し,次
        out     mocw2,al        ]の割込みを受け付けるようにする
        pop     bx
        pop     ax
        pop     ds
        iret ——————————————————— IRET ではフラグも復帰するため
                                 STI はこの場合不要
stop_spool:
        assume  ds:system_seg
        in      al,mimr         ]
        or      al,04h          ]8259A の CRTV をマスク
        out     mocw1,al        ]
        cli
        mov     ax,system_seg   ]
        mov     ds,ax           ]
        mov     ax,cs:save_seg  ]
        mov     crtv_seg,ax     ]保存しておいたベクタを元に戻す
        mov     ax,cs:save_adr  ]
        mov     crtv_adr,ax     ]
        sti                     ]
        jmp     busy ——————————— busy へ飛んで,処理を終了する
;
main    ends
        end
```

リスト 7-2-3 起動プログラム

```
100 ' print utility 1/2 by m.f.
110 CLEAR ,&H1B00: DEFINT A-T: CLS 3: SCREEN ,,,0
120 DEF SEG=&HA800: VAD=4
130 INPUT "file name";F$
140 OPEN F$ FOR INPUT AS #1
150 WHILE NOT EOF(1)
160   LINE INPUT #1,A$: A$=A$+CHR$(&HD)
170   FOR I=1 TO LEN(A$)
180     POKE VAD,ASC(MID$(A$,I,1))
190     VAD=VAD+1
200   NEXT
210 WEND
220 POKE 1,VAD¥256: POKE 0,VAD MOD 256
230 DEF SEG=&H1B00: SPOOL=0
240 BLOAD "L7-2-2"
250 CALL SPOOL
260 END
```

8章
マシン語ランダム・テクニック

この章では，今まで解説してきたPC-9801のハードウェアや8086のマシン語プログラミングの基礎技術を応用した，いくつかのプログラムを紹介します．市販のパッケージ・ソフトとは異なり，ひとつひとつのプログラムを単独で実用とすることはできませんが，様々なプログラム・テクニックが挿入されていますので，リストを根気よく解析すれば，PC-9801でのマシン語による応用プログラムを作成する際の，よい参考になると思います．

なお，ここに掲載したマシン語プログラムは，すべてBASICから利用することを考慮しているため，MS-DOSを使ってマシン語を入力した場合は，巻末の転送ルーチンで，アセンブルしてできたBINファイルをBASICディスクに転送したのち，ロードして使用してください．

8-1 キー入力ルーチン

8-1-1 キーボード・コミュニケーション

PC-9801は，本体と分離したキーボード・ユニットとシリアル通信（調歩同期方式）でキーデータの受渡しを行っています．

キーボードからのシリアルデータは，本体内のμPD8251Aによって8ビットのパラレル・データに変換されて内部で処理されます．このデータはキーを押した時と離した時に送られ，データの最上位ビット（ビット7）をこの区別に，残りの7ビットを，どのキーが押されたかを表すのに使います．8251Aはデータを受信すると，割込みコントローラの8259Aに割込み要求を出しますので，1回キーを押して離すと，8259Aからは，2回の割込み要求をCPUに出すことになります．

キーとデータの関係は図8-1-1のようになっており，たとえばテンキーの"1"を押すと4AHが送られ，離すとCAHが送られてきます．

8-1-2 キーデータを取り込む

マシン語プログラムでキー入力を処理するには，BIOSのルーチンを用いる方法，キーデータ処理用の8251Aの状態をプログラムで直接調べる方法，キーボードからの割込みを使って処理する

方法などが考えられます．

このうちBIOSを利用すれば，コード変換の手間が省けて便利ですが，キー入力に対する速い対応を行うには不安があります．あとの2つの方法は，図8-1-1のコード表からアスキーコード

押した時の出力データ

					D7	0	0	0	0	0	0	0	0
					D6	0	0	0	0	1	1	1	1
					D5	0	0	1	1	0	0	1	1
					D4	0	1	0	1	0	1	0	1
D3	D2	D1	D0			0	1	2	3	4	5	6	7
0	0	0	0	0		ESC	Q タ	F ハ	< , ネ .	−	.	STOP	SHIFT
0	0	0	1	1		1 ! ヌ	W テ	G キ	> . ル °	/		COPY	CAPS
0	0	1	0	2		2 " フ	E イ ィ	H ク	/ ? メ ・	7		f・1	カナ
0	0	1	1	3		3 # ア ァ	R ス	J マ	− ロ	8		f・2	GRPH
0	1	0	0	4		4 $ ウ ゥ	T カ	K ノ	△	9		f・3	CTRL
0	1	0	1	5		5 % エ ェ	Y ン	L リ	XFER	*		f・4	
0	1	1	0	6		6 & オ ォ	U ナ	; + レ	ROLL UP	4		f・5	
0	1	1	1	7		7 ' ヤ ャ	I ニ	: * ケ	ROLL DOWN	5		f・6	
1	0	0	0	8		8 (ユ ュ	O ラ	] | ム 」	INS	6		f・7	
1	0	0	1	9		9) ヨ ョ	P セ	Z ツ ッ	DEL	+		f・8	
1	0	1	0	A		0 ワ ヲ	@ ~ ゛	X サ	・	1		f・9	
1	0	1	1	B		− = ホ	[| 「 ゜	C ソ	←	2		f・10	
1	1	0	0	C		^ ヘ	⏎	V ヒ	→	3			
1	1	0	1	D		¥ | −	A チ	B コ	↓	=			
1	1	1	0	E		BS	S ト	N ミ	HOME CLR	0			
1	1	1	1	F		TAB	D シ	M モ	HELP	,			
D3	D2	D1	D0			8	9	A	B	C	D	E	F
					D4	0	1	0	1	0	1	0	1
					D5	0	0	1	1	0	0	1	1
					D6	0	0	0	0	1	1	1	1
					D7	1	1	1	1	1	1	1	1

離した時の出力データ

図8-1-1　キーコード表

等を得るための処理を必要に応じて作らなければなりませんが，使用するキーを限定したりすると，速い対応ができます．とくに割込みを使えば，無駄な入力待ちにCPUの実行時間を費やされることなく効率良い入力ができますので，ゲームなどで，キー入力によってリアルタイムな処理を行いたい場合などにはうってつけです．

ここではキー入力ルーチンの例として，割込みを用いて，8個まで同時入力データを保持するルーチンを，リスト8-1-1に示します．キーテーブルの8バイトの領域にキーボードから送られてきたデータを格納していくだけの簡単な例ですが，このテーブルを走査して，同時に押されているキーを判別したり，キーを離した時の割込みで，そのキーデータを有効にするというように改造して使って下さい．テーブルを参照する際には，割込みがかかってデータが書き変わることのないよう，割込み許可フラグをリセットするようにします．

リスト 8-1-1　キー入力サブルーチン

```
;       key input sample 1/20 by m.f.
;
mocw2                   equ     00h      ――マスタ 8259A のOCW2コマンド用ポート・アドレス
key_data                equ     41h      ┐
key_status              equ     43h      │キーボード用8251Aのポート・アドレス
key_command             equ     43h      ┘
;
system_seg              segment at 0     ┐
                        org     9*4      │
keyint_off              dw      ?        │キーボード用8251Aの割込みベクタ格納アドレス
keyint_seg              dw      ?        │
system_seg              ends             ┘
;
sub1    segment
        assume  cs:sub1,ds:system_seg
;
key_init:
        push    ax
        push    bx
        push    cx
        push    ds
        cli
        mov     ax,system_seg                 ┐
        mov     ds,ax          実際に使用する  │
        mov     ax,sub1――――セグメント・ア   │
        xchg    ax,keyint_seg  ドレスを入れる  │キー割込みのベクタを書き換える
        mov     cs:save_seg,ax                │元のベクタは保存しておく
        mov     ax,offset keyint_entry        │
        xchg    ax,keyint_off                 │
        mov     cs:save_off,ax                ┘
;
        mov     cx,8                 ┐
        mov     al,80h               │
        mov     bx,offset key_table  │キーデータが格納される領域のクリア
l1:     mov     cs:[bx],al           │MSBがセットされているとそこにデータをいれ
        inc     bx                   │ることができる
        loop    l1                   ┘
        pop     ds
        pop     cx
```

```
        pop     bx
        pop     ax
        sti
;
        ret
;
save_seg        dw      ?          元のベクタが格納される領域
save_off        dw      ?
;
keyint_entry:                      キー割込みのエントリ
        push    ax
        push    bx
        push    dx
        push    di
;
        mul     ax                 意味のない命令．キーデータをリードするまでに
                                   15μs以上必要なため
        in      al,key_status      ステータスを読み，エラーをチェック
        and     al,38h
        mov     al,16h             RTY(リトライなし)コマンドを出す
        out     key_command,al
        in      al,key_data        キーデータのリード．これによりRDYは
                                   クリアされる
        jz      next
        mov     bx,offset key_table
        mov     cx,8
        mov     dl,al              キーデータをDLに保存
        or      al,al
        jns     make1              SFがセットされていればキーが押された時の割込み
        and     al,7fh
break1:                            キーを離した時の割込み処理
        cmp     al,cs:[bx]         データ領域を走査して同じデータでMSBがリセ
        jz      next1              ットされているものがあれば，それを書き換えて
        inc     bx                 つまりMSBをセットする
        loop    break1
        jmp     short next
make1:
        test    cs:[bx],byte ptr 80h   キーを押した時の割込み処理
        jnz     next1                  データ領域を走査し，MSBがセットされている
        inc     bx                     ところがあれば，そこにデータを書き込む
        loop    make1
        jmp     short next
;
next1:  mov     cs:[bx],dl
;
next:
        mov     al,20h             8259AにEOIコマンドを出す
        out     mocw2,al
        pop     di
        pop     dx
        pop     bx
        pop     ax
;
        iret
;
key_table       db      8 dup(?)   キーのデータが書き込まれる領域
                                   別のルーチンでここを調べた時MSBがリセット
                                   されたデータがあればそれはそのキーが押されて
                                   いることを表している
;
sub1    ends
        end
```

8-1-3 キー入力によりグラフィック・パターンを動かす

リスト8-1-2は16×16ビットの大きさのパターンをグラフィック画面に表示して，それをキー入力によって8方向に動かすプログラムです．パターンを動かすには，新たなパターンの書込みと，古いパターンの消去をしなければなりません．この例ではXOR命令を使い，ビット反転をして，これを行っています（リスト8-1-2）．

リスト 8-1-2 パターン移動ルーチン

```
;          key input sample 1/19 by m.f.
;
mocw2              equ       00h      ┐
mimr               equ       02h      │マスタ8259Aのポート・アドレス
mocw1              equ       02h      ┘
crtres             equ       64h ———— GDCのリセットコマンド用ポート・アドレス
key_data           equ       41h      ┐
key_status         equ       43h      │キーボード用8251Aのポート・アドレス
key_command        equ       43h      ┘
;
main_seg           equ       1b00h ——— このプログラムのロード・アドレス
;
d_length           equ       16 ————— 出力パターンのビット長
maxy               equ       400      ┐X,Y方向の座標
maxx               equ       640      ┘
speed              equ       2 —————— 一度のCRTV割込みで2ビットずつ移動する
;
system_seg         segment at 0       ┐
                   org       9*4      │
keyint_off         label     word     │
                   org       9*4+2    │
keyint_seg         label     word     │キー割込みとCRTV割込みの
                   org       10*4     │ベクタ格納アドレス
crtv_adr           label     word     │
                   org       10*4+2   │
crtv_seg           label     word     │
system_seg         ends               ┘
;
main      segment
          assume   cs:main,ds:system_seg
          mov      ax,0               ┐
          mov      cs:xdata,ax        │パターンを座標(0,0)から表示する
          mov      cs:ydata,ax        ┘
          call     crtv2
          call     key_init ———————— キーボード割込みのイニシャライズ
          call     crtv_init ——————— CRTV割込みのイニシャライズ
dummy:    jmp      short dummy ————— 無限ループ，メインルーチンはこれで終り，あと
                                     の処理は割込み処理で行う
;         ————————————————————————— CRTV割込みエントリ
crtv_entry:
          assume   ds:main
          push     ds
          push     es
          push     ax
          push     bx
          push     cx
          push     dx
          push     si
          push     di
```

```
        call    crtv2           ; 現在表示しているパターンを消す
        mov     ax,cs
        mov     ds,ax
        mov     al,c_data
        rcr     al,1
        jnc     n1
        add     xdata,speed
        cmp     xdata,maxx-d_length
        jc      n1
        mov     xdata,maxx-d_length-1
n1:     rcr     al,1
        jnc     n2
        add     ydata,speed
        cmp     ydata,maxy-d_length
        jc      n2
        mov     ydata,maxy-d_length-1
n2:     rcr     al,1
        jnc     n3
        sub     ydata,speed
        jns     n3
        mov     ydata,0
n3:     rcr     al,1
        jnc     n4
        sub     xdata,speed
        jns     n4
        mov     xdata,0
n4:
;
        call    crtv2           ; 座標変更後のパターン表示
        mov     al,20h
        out     mocw2,al
        out     crtres,al       ; GDCにリセット・コマンドを送る
        pop     di
        pop     si
        pop     dx
        pop     cx
        pop     bx
        pop     ax
        pop     es
        pop     ds
        sti
        iret
;
;
crtv2:
        assume  ds:main
        mov     ax,cs
        mov     ds,ax
        mov     ax,ydata
        add     ax,ax
        add     ax,ax
        add     ax,ax
        add     ax,ax
        mov     dx,ax
        add     ax,ax
        add     ax,ax
        add     dx,ax
;
        mov     ax,xdata
        mov     bx,ax
```

Annotations:

- mov al,c_data … mov xdata,0: c＿dataを調べ，押されているカーソルキーの方向に応じてパターンの表示座標を変更する 座標の境界を調べ端まで行くと，それ以上その方向にはいかないようにしている
- mov al,20h / out mocw2,al: 8259AにEOIコマンドを送る
- crtv2: xdata，ydata の値から実際のV-RAMアドレスを計算しパターンを表示するルーチン
- mov ax,ydata … mov bx,ax: V-RAMアドレス=ydata＊80+(xdata¥8) ビット位置=xdata mod 8

```
        mov     cl,3
        shr     ax,cl
        add     ax,dx
        mov     adres,ax
        and     bx,7
        mov     bit,bx
;
        mov     ax,0a800h                 V-RAMセグメントをesに設定
        mov     es,ax
        mov     si,offset d_table         パターンデータの先頭アドレスを設定
        mov     di,adres
        mov     cx,d_length               パターンの縦のビット長
w1:     push    cx
        mov     bl,0
        mov     ax,[si]
        mov     cx,bit
        jcxz    w2                        横方向(X方向)へのデータの書き込み
w3:     shr     ax,1                      パターンの横ビット長は16なので2バイトまたは3
        rcr     bl,1                      バイトの書き込み(XOR)を行う
        loop    w3
        xor     es:[di+2],bl
w2:     xchg    al,ah
        xor     es:[di],ax
        inc     si                        パターンデータのポインタをインクリメントする
        inc     si
        add     di,maxx/8                 縦方向にはV-RAMアドレスに80を加える
        pop     cx
        loop    w1
        ret
;
xdata   dw      ?                         表示しているパターン左上の座標
ydata   dw      ?
;
bit     dw      ?                         座標の実アドレスとビット位置
adres   dw      ?
;
d_table label   word
        dw      0000001111000000b
        dw      0000111111110000b
        dw      0011111111111100b
        dw      0011111111111100b
        dw      0111111111111110b
        dw      0111111111111110b
        dw      1111111111100011b
        dw      1111111111100011b         パターンのデータ
        dw      1111111111100011b
        dw      1111111111100011b
        dw      0111111111111110b
        dw      0111111111111110b
        dw      0011111111111100b
        dw      0011111111111100b
        dw      0000111111110000b
        dw      0000001111000000b
;
;
crtv_init:                                CRTV割込みのイニシャライズルーチン
        assume  ds:system_seg
        push    ds
        mov     ax,system_seg
        mov     ds,ax
```

```
        cli
        mov     ax,main_seg                    ; ベクタ・アドレスの書き換え
        xchg    ax,crtv_seg                    ; 元のアドレスは保存しておく
        mov     cs:scrtv_seg,ax
        mov     ax,offset crtv_entry
        xchg    ax,crtv_adr
        mov     cs:scrtv_adr,ax
;
        in      al,mimr                        ; CRTV割込みマスクの解除
        and     al,0fbh
        out     mocw1,al
        out     crtres,al ---------------------- GDCにリセット・コマンドを出す
        pop     ds
        sti
        ret
;
scrtv_seg       dw      ?                      ; 元のベクタ・アドレスの退避場所
scrtv_adr       dw      ?
;
;
key_init: -------------------------------------- キーボード割込みのイニシャライズ
        push    ax
        push    ds
        push    cx
        push    bx
        cli
        mov     ax,system_seg                  ; ベクタの書き換えを行う
        mov     ds,ax
        mov     ax,main_seg
        xchg    ax,keyint_seg
        mov     cs:save_seg,ax
        mov     ax,offset keyint_entry
        xchg    ax,keyint_off
        mov     cs:save_off,ax
;
        pop     bx
        pop     cx
        pop     ds
        pop     ax
        sti
;
        ret
;
save_seg        dw      ?                      ; 元のベクタ・アドレスの退避場所
save_off        dw      ?
c_data          db      0 ---------------------- 押されたカーソルキーの方向にデータが下位4ビットにセットされる
;
keyint_entry:
        push    ax
        push    bx
        push    cx
        push    dx
        push    di
;
        mul     ax                             ; ステータスを読む前に少しwaitをおくための意味のない命令
        mul     ax
        in      al,key_status                  ; ステータスを読むことにより8251AのRDY(上線付き)を解除する
        and     al,38h
        mov     al,16h                         ; データ転送リトライを行わない
        out     key_command,al
```

```
        in      al,key_data
        jnz     next
        mov     dl,al
        and     al,7fh
        cmp     al,3dh+1
        jnc     next
        cmp     al,3ah
        jc      next
        and     al,3
        mov     cl,al
        mov     al,1
        clc
        rcl     al,cl
        test    dl,80h
        jz      make1
break1: not     al
        and     cs:c_data,al
        jmp     short next
make1:  or      cs:c_data,al
;
next:
        mov     al,20h
        out     mocw2,al
        pop     di
        pop     dx
        pop     cx
        pop     bx
        pop     ax
;
        iret
;
main    ends
        end
```

キーデータを読む (in al,key_data)

カーソルキーのデータかを判定し，方向に応じて4ビット(4方向)のうちの1ビットをセットしたデータを得る (and al,7fh ～ jz make1)

カーソルキーを離した時の割込みであればそのビットをクリアする (break1: ～ jmp short next)

カーソルキーを押した時の割込みであればそのビットをセットする (make1: or cs:c_data,al)

8259AにEOIコマンドを送る (mov al,20h / out mocw2,al)

リスト 8-1-3　起動プログラム

```
100 SCREEN 2,0,1: CLS
110 DEF SEG=&H1B00
120 BLOAD "L8-1-2"
130 A=0: CALL A
140 END
```

リスト8-1-2はリスト8-1-3を使って起動します．

キー入力はカーソルキーのみを調べていて，4ビットの方向データを得ています．これをパターン移動ルーチンで参照して，4方向に1ビットずつパターンを動かしますので，上下方向と左右方向のキーを合成して斜め方向のデータを得ることで，8方向への移動を行います．

8-2 行列演算ルーチン

8-2-1 マトリクス演算プログラム

行列（マトリクス）の演算は数学だけでなく，物理学や工学，経済学などにまで広く利用されています．コンピュータの世界でも，最近はやりの CG の基礎である座標変換などは，行列によって行う場合が多いようです．

3 次元の座標変換などでは，4×4（4 行，4 列）の行列が多用されるので，ここでは，2 バイトの整数値（−32768〜32767）を要素とする 2 つの 4×4 行列の和，差，積（結合）を求めるプログラムを紹介しましょう．

リスト 8-2-1 の BASIC プログラムは，入力された行列の要素を，2 次元配列 A，B に代入し，演算の種類を指定してマシン語ルーチンを CALL するものです．

CALL 文の引数には，配列 A，B それぞれの先頭アドレスと，結果が格納される配列 C の先頭アドレス，および，演算の種類を示す変数 FUNC を与えています．

リスト 8-2-2 のマシン語ルーチンは，与えられた A，B，2 つの配列のデータを，FUNC の値に従って和，差，積のどれかを求めるルーチンを呼んで処理したうえ，配列 C の内容を直接書き変えて，結果を格納します．

リスト 8-2-1 データ入力部

```
100 ' -----------------------------------------------------------
110 '            4*4 matrix       add , sub , mul
120 ' -----------------------------------------------------------
130 CLEAR,&H1B00 : DEFINT A-Z : DEF SEG=&H1B00 : BLOAD "mat4.mon"
140 DIM A(3,3),B(3,3),C(3,3) : MAT=0
150 '
160 MATRIX$="A" : GOSUB *INPUT.M
170 MATRIX$="B" : GOSUB *INPUT.M
180 '
190 PRINT " Which do you select ?"
200 INPUT "   1: add  2: sub  3: mul  ",FUNC
210 FUNC=FUNC-1
220 '
230 CLS
240 CALL MAT(A(0,0),B(0,0),C(0,0),FUNC)
250 PRINT HEX$(FUNC/256);" ... if there is a error then FE"
260 '
270 PRINT
280 MATRIX$="A" : GOSUB *PRINTOUT
290 MATRIX$="B" : GOSUB *PRINTOUT
300 MATRIX$="C" : GOSUB *PRINTOUT
310 '
320 END
330 '
340 *INPUT.M
350 FOR I=0 TO 3
360   FOR J=0 TO 3
370     IF MATRIX$="A" THEN PRINT "A(";I;",";J;")= ";:INPUT A(I,J)
380     IF MATRIX$="B" THEN PRINT "B(";I;",";J;")= ";:INPUT B(I,J)
```

```
390    NEXT J
400 NEXT I
410 RETURN
420 '
430 *PRINTOUT
440 PRINT "MATRIX ";MATRIX$
450 FOR I=0 TO 3
460   FOR J=0 TO 3
470     IF MATRIX$="A" THEN PRINT USING "#######";A(I,J);
480     IF MATRIX$="B" THEN PRINT USING "#######";B(I,J);
490     IF MATRIX$="C" THEN PRINT USING "#######";C(I,J);
500   NEXT J
510   PRINT
520 NEXT I
530 PRINT
540 RETURN
```

リスト 8-2-2 行列演算ルーチン

```
;         mat function    ( 4*4 matrix )
;         2/10
;   usage: call mat(a%(0,0),b%(0,0),c%(0,0),func)
;   func: add=0,sub=1,mul=2
;   return func: func _overflow=0feh (high 8bit)
;
param_a equ       0ch
param_b equ       08h
param_c equ       04h
;
main      segment
          assume cs:main,ds:main
          push    ds
          push    bx
          cld
          les     si,[bx]                      ┐ funcの値をDLに格納
          mov     dl,es:[si]                   ┘
          mov     es:[si+1],byte ptr 0         ── funcの上位をクリア
          mov     cs:err_f,0                   ── エラーフラグを初期化
;
          cmp     dl,2                         ┐
          jz      to_mul                       │ DL=0, 1のときaddsubをcall
          call    addsub                       │ DL=2のときmulをcall
          jmp     p_end                        │
to_mul:   call    mul                          ┘
;
p_end:    mov     ah,cs:err_f                  ┐
          pop     bx                           │
          pop     ds                           │ funcの上位にエラーフラグの値を返す
          les     si,[bx]                      │
          mov     es:[si+1],ah                 ┘
          iret                                 ── BASICに復帰
;
addsub:
          les     si,[bx+param_a]              ── A(0,0)のアドレスのセグメントをESに
                                                  オフセットをSIに格納する
          mov     bp,32                        ┐
loop1:                                         │ BPがカウンタ,BP+SIがポインタで,
          dec     bp                           │ A(3,3)から順にA(2,3), ……, とA(0,0)
          dec     bp                           │ までスタックに格納する
```

```
        push    es:[bp+si]
        jnz     loop1
;
        les     di,[bx+param_c]   ——— C(0,0)のアドレスのセグメントをES，オフセットをDIに格納する
        lds     si,[bx+param_b]   ——— B(0,0)のアドレスのセグメントをDS，オフセットをSIに格納する
        mov     cx,16   ;4*4 mat  ——— CXがloop2のカウンタ(4×4行列の成分は16コ)
loop2:
        pop     ax
        cmp     dl,1
        jz      sub
add:    add     ax,[si]
        jmp     errchk
sub:    sub     ax,[si]
                                  ] DL=0のとき減算
                                    DL=1のとき加算
;
errchk: jno     store   ;not ov
        mov     cs:err_f,0feh
                                  ] オーバーフローのとき，CS：err_fにFEHを代入．
                                    エラーでないときstoreへ
store:  stosw
        inc     si
        inc     si
        loop    loop2
                                  ] AXに格納された行列Aと行列Bの成分の和または差を
                                    ES：DIの示す行列Cの成分のアドレスに格納する
        ret                       ——— メインルーチンへ戻る
;
mul:
        les     si,[bx+param_a]
        mov     cs:asi,si
        mov     cs:aes,es
                                  ] A(0,0)の格納アドレスのセグメントをCS：aesに，オフセ
                                    ットをCS：asiに格納する
        les     si,[bx+param_b]
        mov     cs:bsi,si
        mov     cs:bes,es
                                  ] B(0,0)の格納アドレスのセグメントをCS：besに，オフセ
                                    ットをCS：bsiに格納する
        lds     di,[bx+param_c]   ——— C(0,0)の格納アドレスのセグメントをDSに，オフセットをDIに格納する
;
        xor     bx,bx
        xor     bp,bp
        xor     cx,cx
                                  ] BX(行列Aの成分のインデックス)，BP(行列Bの成分の
                                    インデックス)，CX(積を格納するレジスタ)の値をクリア
m10:    mov     es,cs:aes
        mov     si,cs:asi
        mov     ax,es:[bx+si]
                                  ] 行列Aの成分の１つをAXに格納する
        mov     es,cs:bes
        mov     si,cs:bsi
        mov     dx,es:[bp+si]
                                  ] 行列Bの成分の１つをDXに格納する
        imul    dx                ——— 行列A，Bの成分の積の上位をDX，下位をAXに格納する
        add     cx,ax
        adc     cs:sum_up,dx
                                  ] 上で求めた積をsum_up：CX(4バイト)に加える
        jno     m11
        mov     cs:err_f,0feh
                                  ] オーバーフローのチェック．エラーなしのときm11へ
m11:
        cmp     bx,24
        jnb     m20
        add     bx,8
        inc     bp
        inc     bp
        jmp     m10
                                  ] 行列Aの１行分の成分をすべて(４つ)掛け終えたときm20へ，
                                    終えていないときは，行列Aの１行横へ，行列Bの１列下へイ
                                    ンデックスを移す
;
m20:
        mov     dx,cs:sum_up
        cmp     dx,0
        jz      m21
        cmp     dx,0ffffh
        jz      m21
        jmp     m22
                                  ] sum_upの各ビットとCXのMSBがすべて０またはすべて
                                    １のとき，sum_up：CXとCXの値が等しくなる．このとき
                                    エラーなし
```

```
m21:
        xor     dx,cx
        jns     m23
m22:
        mov     cs:err_f,0feh ── エラーフラグをたてる
m23:
        mov     ds:[di],cx   ── CXの値を行列Cの成分として格納する
        xor     cx,cx        ┐
        inc     di           │ CXをクリアし,行列Cの成分のインデックスであるdiを2だけ増す
        inc     di           ┘
        cmp     bx,30        ┐ 行列Bのある1列に対して,行列Aのすべての行を掛け終えたかをチェッ
        jz      m30          ┘ クする. 終えていればm30へ
        sub     bx,22        ┐
        sub     bp,6         │ 行列Aの次の行へ, 行列Bの現在演算中の列のはじめの成分のインデック
        jmp     m10          ┘ スを移し, m10へ飛んで計算をつづける
m30:
        cmp     bp,30        ┐ 行列Bのすべての列を計算し終えていればm40へ移って積の演算を
        jz      m40          ┘ 終了する
        inc     bp           ┐
        inc     bp           │ 行列Aの1番目の成分A(0,0)へ, 行列Bの次の列へインデックスを移し,
        xor     bx,bx        │ m10へ飛んで計算を続ける
        jmp     m10          ┘
m40:
        ret     ──────────── メインルーチンへ戻る
;
err_f   db      ? ────────── エラーフラグの値が格納される
;
asi     dw      ?            ┐
bsi     dw      ?            │ aes, asiにはA(0,0)の格納アドレスのセグメントとオフセットが,
aes     dw      ?            │ bes, bsiにはB(0,0)の格納アドレスのセグメントとオフセットが
bes     dw      ?            ┘ それぞれ格納される
sum_up  dw      0 ────────── 積の計算の途中での積の和の上位2バイトが格納される
;
main    ends
        end
```

8-2-2 プログラムの実行と注意点

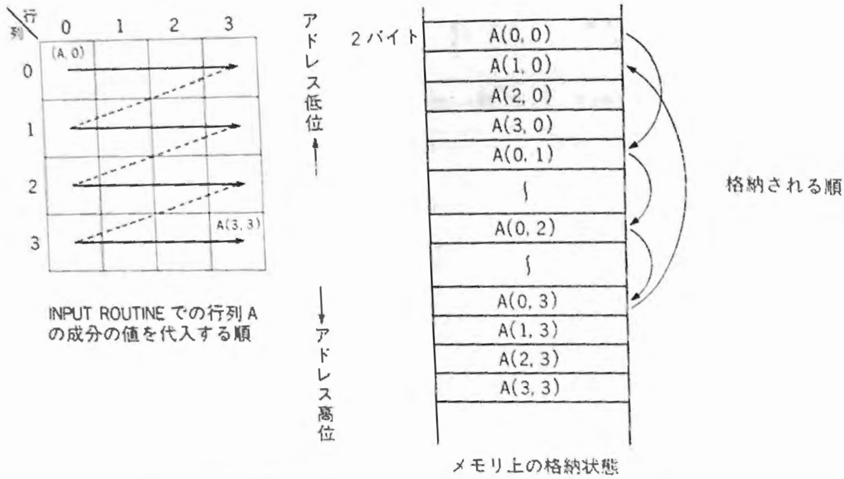
まず，リスト8-2-2のアセンブラ・プログラムを入力し，アセンブラ等で出力されたオブジェクト（MASMならBINファイル）をBASICのディスクに〝mat4 . mon″というファイル名で格納しておきます．

つぎに，BASICプログラムを入力して実行すると，ディスクに格納されたマシン語ルーチンがDEF SEGで指定されたセグメント（1B00H）の0番地からロードされます．

BASICプログラムでは，続いて，行列A，Bの順で要素の値を聞いてきますので入力してください．

要素の入力は図8-2-1に示した順に行います．

データの入力が終ると，演算の種類の入力を促しますので，1～3の数字で入力してください．すると，前でロードしたマシン語ルーチンがCALLされ，演算処理を行ったのち結果が画面に出力されます．



※N_{88}-BASICの配列の取り方により，インプットルーチンでの行列の成分の値の代入はメモリ上の連続したアドレスには代入されていかない．そのため，マシン語ルーチンの行列成分とアドレス指定の対応がわかりにくくなっている

図8-2-1　行列成分の代入する順序とメモリの対応

8-2-3　マシン語プログラムの構造

マシン語プログラムは，

① BASIC の CALL 命令の引数の取込みとエラーチェック，および BASIC への復帰処理をする部分

② 加・減算，および配列 C への結果の格納部分

③ 乗算，および結果格納部分

に大きく分けられます．

①は，ラベル main からラベル p_end までの部分で，ここでは，まず最初に BASIC とのデータリンクに使用されるレジスタ DS，BX を PUSH します．

つぎに，BASIC の CALL 命令の実行により CALL 文の引数テーブル (引数の実際の値が入っているアドレスのセグメントとオフセットや，最後尾の引数から逆順に格納されている）の先頭アドレスのセグメント値とオフセット値が DS, BX に入るので，これをもとに LES 命令で最後尾の変数，FUNC の値のあるアドレスを求めます．

LES 命令は，第 4 章でも出てきましたが，ソース部に置かれた 16 ビット・レジスタの値をもとに，この値の示すアドレスと次のアドレスを下位，上位の順に合せて 16 ビットデータとし，ディスティネーション部に置かれた 16 ビット・レジスタに格納したうえ，さらに 2 つ先のアドレスと 3 つ先のアドレスを下位，上位の順に合せて ES レジスタに格納するものです．

続いて，FUNCの値を求め，各演算ルーチンに処理を渡します．

エラー処理は，エラー(オーバーフロー)が起きた時にFUNCにその値(オーバーフロー＝FE00H)を返してBASICに戻ることで行います．

②の加・減算処理部はラベルaddsubからstoreまでの部分で，DS，BXの値から配列Aと配列Bの各データの入っているアドレスを求め，ループを組んで次々にデータの加算または減算を行います．

このループ中に，STOSWという命令がありますが，これは，AXの値をDIの示すアドレスとその次のアドレスに下位，上位の順で格納したうえ，DIの値を2増やす命令で，この場合は，配列Cの中に演算結果を格納する作業を行います．

③の乗算処理部は，ラベルmul以下の部分で，行列の積(結合)の手順に従い，乗算を行います．

リスト中にコメントを付けておきましたので，詳しくはそちらを見てください．

行列計算の手順は，数学の本にその解説をまかせて，ここでは触れないことにします．

8-3　RAMディスク

8-3-1　RAMディスクとは

フロッピーディスクは現在パソコンで用いられる最もポピュラーな外部記憶装置で，PC-9801にもいくつかのタイプが用意されています．これらは従来のオーディオ・カセットテープによるデータ記憶装置に比べ，信頼性の高い，高速アクセスを可能にしてくれました．しかし，この便利さにも慣れてくると，もっと高速な，大容量の装置を欲するようになってきます．

RAMディスクは，外部記憶(データ保存)装置としての役目をはたすことはできませんが，高速という点ですばらしい性能を示すディスクで，最近の半導体メモリの低価格化により，これからますます使われるようになると考えられます．

機械駆動部分をもったディスクの遅さをおぎなうために，今までにも，ディスクとのデータの入出力のための高速バッファとして高速メモリを用いたキャッシュと呼ばれる方法を使用することは行われていましたが，RAMディスクはディスクそのものを肩代りするという意味でキャッシュとは区別されます．

RAMディスクを実現するには，外部にメモリ装置を接続して，ハードウェアで構成する方法もありますが，ここでは本体内のRAMを使い，ソフトウェアでこれを行うことを考えてみます．

基本的な方針としては，物理的なディスク装置を制御している部分を書き換えて，トラックやセクタを本体内のメモリ上に展開することにします．ただしBASICから通常のディスク装置と同じように扱えるようにするためには，基本的な入出力ルーチンだけでなく，より上位の処理ルーチンとの対応を考慮しなければなりません．

8-3-2 ディスクLIO

第6章で述べたとおり，PC-9801のBASIC処理系は，入出力に関する部分をBIOS，LIOというように階層化して分担処理しています．ディスクLIOには，図8-3-1に示すようなコマンドがあり，BASICのディスク関連命令はLIOの各コマンドに展開されて実行されます．LIOレベルでは，BIOSのコマンドのように，トラック，セクタなどの物理的な値ではなく，FCB(File Control Block)と呼ばれるデータテーブルに書かれたファイル名やレコードという単位でのアクセスが行なわれます．

LIOコマンド	処　　理
OPEN	指定ファイルをオープンする．GET,PUTを行う前に実行すること
CLOSE	指定ファイルをクローズする．FATが更新されていれば書戻しを行う
SGET	ディレクトリを256バイト単位で読み込み，必要なファイルをサーチしてFCBを作成する
SREP	SGETで読み込まれたディレクトリブロックを書き戻す
SDEL	指定ファイルを削除する．ディレクトリ，FATの更新も行われる
GET	指定ファイルのデータを256バイト単位で読み込む
PUT	指定ファイルのデータを256バイト単位で書き込む
PIO	指定デバイス上のデータに対して物理入出力を行う
SENSE	指定されたドライブの状態・媒体の識別を行う

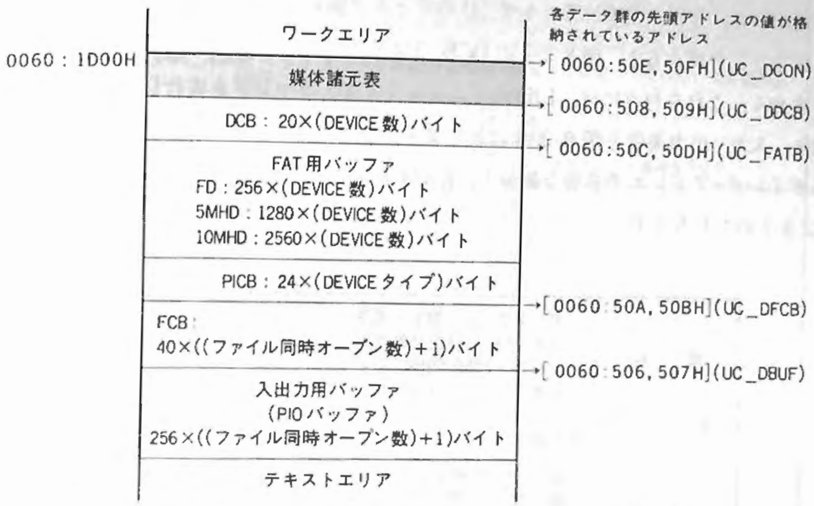
図8-3-1　ディスクLIOのコマンド表

ディスクLIOのコマンドはRAMディスク・プログラムでは使いませんが，LIOのワークエリアには，BASIC処理系に，現在どのような容量のディスクが何台接続されているか，というような情報が書かれています．図8-3-2は，第6章でも一部説明したテキスト・セグメント内のワークエリアです．このうちオフセット1D00Hからは媒体諸元表というディスクに関するパラメータがおさめられています．1つのディスクタイプにつき24バイトがとられ，その意味は図8-3-3に示すとおりです．

このテーブルを書き換えることによりディスク・アクセスに制限を加えることができます．たとえば8インチ2Dのディスクを使っている時，このテーブルのトラックあたりのセクタ数を26(001AH)から25(0019H)に変更すると，セクタ26はBASICのディスク操作命令で読み書き不能になります．

またFATの位置などを変更すれば通常のBASICディスクのアクセスが不能になります．

BASICでRAMディスクが使えるようにするには，このテーブルに，あたかも，もう1台のディスクがあるかのように，パラメータをセットしなければなりません．このテーブルはLIOのイニシャライズで，ディスク装置の接続状況に応じて作成されるものなので，すでに登録されている装置のうち，使用していない部分を書きかえてセットします．現在ほとんど使用されていない



※[ ]内のアドレスと次のアドレスにその位置を示すデータがはいっている

図8-3-2　LIOのワークエリア

媒体種 / パラメータ値	8インチFD	8インチFD / 5インチHD	5インチFD		5インチHD(ハードディスク)		5インチFD	RAM DISK
	1D	2D	1D	2D	5MB	10MB	2DD	RAM
媒体種別	9000H	9004H	5000H	5004H	1台目 8000H 2台目 8004H		7004H	5000H
最大トラック番号(0から)	4CH(76)	4CH(76)	22H(34)	27H(39)	98H(152)	135H(309)	4FH(79)	17H(23)
セクタ／トラック	1AH(26)	1AH(26)	10H(16)	10H(16)	21H(33)	21H(33)	10H(16)	10H(16)
サーフェイス数(0から)	0	1	0	1	3	3	1	0
クラスタ／トラック	1	1	2	2	1	1	1	2
クラスタ／ボリューム	4DH(77)	9AH(154)	46H(70)	A0H(160)	264H(612)	4D8H(1240)	A0H(160)	30H(48)
ディレクトリ／FATのあるトラック番号(サーフェイス0)	23H(35)	23H(35)	12H(18)	12H(18)	46H(70)	96H(150)	28H(40)	0
セクタ／クラスタ	1AH(26)	1AH(26)	08H(8)	08H(8)	21H(33)	21H(33)	10H(16)	08H(8)
FATの開始セクタ番号	15H(21)	18H(24)	0EH(14)	0EH(14)	02H(2)	02H(2)	0EH(14)	0EH(14)
FATの最終セクタ番号	1AH(26)	1AH(26)	10H(16)	10H(16)	21H(33)	21H(33)	10H(16)	10H(16)
FATのコピー数	3	3	3	3	1	1	3	3
IDのあるセクタ	13H(19)	17H(23)	0DH(13)	0DH(13)	1	1	0DH(13)	0DH(13)

※各データは，2バイトデータとして書き込まれている．1つの媒体のデータが2×12＝24バイトを占有する

図8-3-3　媒体諸元表(BASIC)

という理由から，今回の例では5インチ1Dのディスク諸元パラメータ部分を使っています．

ディスクが1台増えると，図8-3-2のDCB，FAT，PICB，の領域それぞれを大きくしなければなりません．これを行うには，LIOのイニシャライズ・コマンドを実行します．このコマンドの機能と，入力，出力条件を図8-3-4に示します．また入力パラメータや，設定されるワークエリアのポインタ・アドレスで必要な部分と，RAMディスク・プログラムで関係するところを図8-3-5にまとめておきます．

機　能	・ディスクLIOに関する部分のイニシャライズを行う ・UCW(図8-3-5)の初期設定 ・バッファ領域の確保(図8-3-2) ・制御ブロックの初期設定
入力条件	内部割込み　INT 0B4H DS：ワーク領域セグメント・アドレス(60H) BP：ワーク領域先頭アドレス(1D00H) UC_DOPN：ファイル同時オープン数(図8-3-5) UC_SRVT：BASIC種別(図8-3-5) N_{88}-BASIC：01H
出力条件	BP：ワーク領域最終アドレス (BP以外は保存)

図8-3-4　LIOイニシャライズ・コマンドの概要

図8-3-2のDCB（Device Control Block）は20バイト単位のデータブロックの集まりで，各ディスクユニット毎に1つのブロックが割り当てられて，そのユニットに関する情報がおさめられています．

FATバッファは，各ディスクユニット毎に読み込んだFATを保存するために使われます．

PICB（Physical I/O Control Block）は，ディスク装置（タイプ）毎に24バイト単位のブロックが割り当てられ，その装置の物理的入出力に関するデータがはいります．

FCB（File Control Block）はBASICレベルのファイルに関するデータが格納される部分で，ファイル名や属性，レコード番号などの情報がおさめられます．各ブロックは40バイトで，ファイルの同時にオープンされる数で領域の大きさが決まります．

プログラム（リスト8-3-1）では，イニシャライズ部分でLIOに関する部分を設定していますが，RAMディスクは，5インチの両面ディスク装置で，これを片面モードとして扱っています．登録処理を簡単にするため，このタイプのディスクの1台目または3台目となるようにしているので，2DDでない5インチディスクは2台までしか同時に使うことはできなくなります．

セグメント	アドレス	ラベル名	バイト数	説明	
0H	55CH	DISK_EQUIP (1)	1	ディスク装置の接続状況(ビットがセットされていると接続) b7 [#3 #2 #1 #0 #3 #2 #1 #0] b0 #3〜#0(上位): 5″FD　#3〜#0(下位): 8″FD　(#はユニット番号を示す)	
	55DH	DISK_EQUIP (2)	1	b7 [#3 #2 #1 #0 0 0 #1 #0] b0 #3〜#0(上位): 5″2DD　#1〜#0(下位): 5″HD	
	560H	DISK_TYPE	1	5″FD装置のタイプ FFH：片面装置 EFH：両面装置 00H：未接続	
	561H	DISK_MODE	1	5″FD両面装置のモード(ビットがセットされていると両面モード) b7 [0 0 0 0 #3 #2 #1 #0] b0 #3〜#0: サーフェイス	
	5CAH	F2DD_MODE	1	5″2DD装置のモード [ ][ ][ ][ ][ ][ ][ ][ ] 上位4ビット: トラック　下位4ビット: サーフェイス トラック：セットされていると80トラックモード リセットされていると40トラックモード サーフェイス：セットされていれば両面モード	
60H	501H	UC_5FD	1	5″FD装置の設置数(最大4台)	UCW
	502H	UC_8FD	1	8″FD装置の設置数(最大4台)	
	503H	UC_DSK	1	ディスク装置の設置合計数(最大12台)	
	504H	UC_DOPN	1	ファイル同時オープン数(0〜15)	
	505H	UC_SRVT	1	BASIC種別(01H：N_{88}-BASIC)	
	506H	UC_DBUF	2	PIOバッファの先頭アドレス	
	508H	UC_DDCB	2	DCB群の先頭アドレス	
	50AH	UC_DFCB	2	FCB群の先頭アドレス	
	50CH	UC_FATB	2	FATバッファの先頭アドレス	
	50EH	UC_DCON	2	媒体諸元テーブルのアドレス	
60H	6A4H	BCTXAD	2	・テキスト格納エリアの先頭アドレス	
	6A6H	BCTXNA	2	・テキストの最終アドレス＋1, この番地の後にデリミタとして0000Hが格納されている ・初期値はBCTXADと同じ	
	6A8H	BCTXEA	2	テキスト格納エリアの最終アドレス＋1	

図8-3-5　RAM DISKに関係するワークエリア

8-3-3　ディスクBIOSコマンド

実際にディスクの入出力を行うのはBIOSの中のルーチンです．BIOSは，INT命令でいくつかのコマンドを実行できるようになっていて，その処理ルーチンはROMに書かれています．ROMのルーチンにはいってしまうとユーザーは手が出せなくなりますから，このプログラムでは，INT命令(INT 1BH)で参照されるベクタ・アドレスを書き換えて別のルーチンへ飛ばし，ここでRAMディスクへのアクセスであれば，それを実現するようにし，そうでなければROM内のルーチン

へ戻しています．つまり，RAM ディスクの BIOS を作りだしているわけです．

5インチ 2D(1D)の BIOS コマンドには，図 **8-3-6** に示すような 7 種類がありますが，ここではそれぞれ図 **8-3-7** に示す処理を行うようにしました．

このRAMディスクは図 8-3-3 のアミ掛けした部分に示す仕様を持ち，V-RAM の A8000H から 96K バイトの大きさとします．このプログラムは少し変更を加えれば，容量を後から設定できるようにすることも可能です．またI/Oバッファの空き領域にこれ自体を転送するようにすれば，メインメモリが自由になります．

	コマンド名	機　　能
1	READ DATA	AL で指定されるデバイスに対して，IDR で指定されるセクタのデータを読み取り主記憶へ転送する(IDRはCL，DH，DL で指定されるパラメータの総称)
2	WRITE DATA	AL で指定されるデバイスに対して，IDR で指定されるセクタに主記憶上のデータを書き込む
3	FORMAT DRIVE	AL で指定されるユニットにマウントされている媒体をフォーマティングする
4	INITIALIZE	AL で指定されるコントローラの初期設定を行う．UA の指定は必要がない．この Initialize を行わないと他のコマンドは起動できない
5	SENSE	AL で指定されるデバイスの状態をステータスとして通知する
6	VERIFY	AL で指定されるデバイスに対して，IDR で指定されるセクタのデータを読み取る．主記憶への転送は行わない
7	SET OPERATION MODE	両面装置に対して Operation Mode(片面アクセス　両面アクセス)を指示する

レジスタ名／コマンド名	AH コマンドコード	AL (50～53H)	BX データサイズ (1～4096)	CL シリンダ番号	DH ヘッド番号 (0～1)	DL セクタ番号 (1～16)	ES データバッファ・セグメント	BP データバッファ・オフセット
リードデータ	06H	○*	○	○	○	○	○	○
ライトデータ	05H	○*	○	○	○	○	○	○
フォーマット	0DH	○*	—	—	—	—	—	—
イニシャライズ	03H	○*	—	—	—	—	—	—
センス	04H	○*	—	—	—	—	—	—
ベリファイ	01H	○*	○	○	○	○	○	○
セット・オペレーション	0EH	○*	—	—	—	—	—	—

※各パラメータをセットして INT 1BH を行う
○の付いているところは該当コマンドにおいて指定が必要なレジスタ．

＊AL はドライブ 1 － 4 を50－53H として扱う．ただし，イニシャライズ，およびセットオペレーションの場合上位 4 ビットは 5 であり，下位 4 ビットは以下の図のようにそれぞれのビットに対し，両面モードで 1，片面モードで 0 となる

ディスク＃1	＃2	＃3	＃4
b_3	b_2	b_1	b_0

図 8-3-6　5インチ2D(1D)BIOS コマンド

コマンド名	説　明
Read Data	指定したトラック，セクタから，それに対応するメモリ・アドレスを計算し，256バイトのデータを読み出す
Write Data	Readと同じように求めたアドレスへデータを書く
Format Drive	ディレクトリ，FAT，ID部分を初期化する
Initialize Sense Verify Set Operation Mode	何もしないで戻る

図8-3-7　RAMディスクのBIOSコマンド

リスト 8-3-1　RAMディスクルーチン　　*PC-9801U/VF/VMでは動作しません

```
;         ram-disk 1/17    by m.f.
;
load_seg          equ      1c00h         ―――――――このプログラムがロードされるセグメント
disk_base         equ      0a800h        ―――――――RAMディスクデータ領域の最初のセグメント
ram_capacity      equ      96            ―――――――RAMディスク容量(Kバイト)
max_track_no      equ      ram_capacity/4-1 ――RAMディスクの最大トラック番号
;
system_seg        segment  at 0          ┐
                  org      6ch           │
intoff            label    word          │INT 1BHのベクタ・アドレスの格納領域
                  org      6eh           │
intseg            label    word          ┘
                  org      55ch          ┐
disk_equip        label    byte          │
                  org      560h          │
disk_type         label    byte          │
                  org      561h          │
disk_mode         label    byte          │
system_seg        ends                   │
;                                        │
text_seg          segment  at 60h        │
                  org      508h          │図8-3-5参照
uc_ddcb           label    word          │
                  org      50eh          │
uc_dcon           label    word          │
                  org      6a4h          │
bctxad            label    word          │
                  org      6a6h          │
bctxna            label    word          │
media_def         equ      1d00h         │
text_seg          ends                   ┘
;
main      segment
          assume  cs:main,ds:system_seg
          org     0
;
;         init ram-disk
          mov     ax,system_seg          ┐
          mov     ds,ax                  │
          mov     bh,00010000b           │
          mov     bl,11111110b           │
```

```
        mov     al,disk_equip
        and     al,0f0h                 5"FD装置の1台目または3台目を追加
        jz      12                      片面モードの選択
        mov     bh,01000000b
        mov     bl,11111011b
12:     or      disk_equip,bh
        and     disk_mode,bl
        cmp     disk_type,0
        jne     11                      未接続であれば両面装置としてセット
        mov     disk_type,0efh
11:     mov     ax,offset ramdisk
        xchg    ax,intoff
        mov     cs:jp_off,ax            INTベクタの書き換えとROM内ルーチンへの
        mov     ax,load_seg             FAR JMPのアドレスをセット
        xchg    ax,intseg
        mov     cs:jp_seg,ax
;
        assume  ds:text_seg
        mov     ax,60h
        mov     ds,ax
        mov     bp,uc_dcon              LIOイニシャライズ
        int     0b4h
;
        mov     bctxad,bp
        mov     bctxna,bp               テキスト・ポインタのイニシャライズ
;
        mov     bx,uc_dcon
14:     cmp     [di],5000h
        je      13
        add     di,24
        cmp     di,uc_ddcb              媒体諸元表中5"1Dの領域をさがす
        jne     14
err:    jmp     short return
;
13:     push    cs
        pop     ds
        mov     ax,text_seg
        mov     es,ax
        mov     si,offset table1        5"1Dの領域にtable 1の内容を転送する
        mov     cx,12
        cld
        rep     movsw
;
        call    f_format
return: iret
;
table1:
        dw      5000h
        dw      max_track_no
        dw      16
        dw      0
        dw      2
        dw      (max_track_no+1)*2      RAMディスクの諸元．図8-3-3参照
        dw      0
        dw      8
        dw      14
        dw      16
        dw      3
        dw      13
;
```

```
;           disk entry
;
ramdisk:
            push        ds
            push        ax
            push        cx
            assume      ds:system_seg
            mov         cx,system_seg
            mov         ds,cx
            mov         ah,al
            and         ah,0f0h
            cmp         ah,50h
            jne         jmp_rom
            and         al,0fh
            mov         ah,disk_equip
            mov         cx,4
15:         shl         ah,1
            jnc         16
            dec         al
16:         loop        15
            cmp         al,0ffh
            je          entry
jmp_rom:pop         cx
            pop         ax
            pop         ds
            db          0eah
jp_off      dw          ?
jp_seg      dw          ?
;
entry:
            pop         cx
            pop         ax
            pop         ds
            sti
            cld
            push        ds
            push        si
            push        di
            push        es
            push        bp
            push        dx
            push        cx
            push        bx
            push        ax
            mov         bp,sp
            sal         ah,1
            mov         bl,ah
            and         bx,001eh
            call        cs:[bx+offset ctable]
exit1:      mov         [bp+1],ah
            and         [bp+16h],byte ptr 0feh
            cmp         ah,20h
            jb          exit
            or          [bp+16h],byte ptr 1
exit:       pop         ax
            pop         bx
            pop         cx
            pop         dx
            pop         bp
            pop         es
```

INT 1BHのBIOSコールの飛び先
RAMディスクへのアクセスでなければROM内ルーチンへのFAR JMPを行う

RAMディスクBIOSのエントリ
ctableのコマンド・アドレスへの分岐を制御する

```
        pop     di
        pop     si
        pop     ds
        iret
ctable:
        dw      none     ;ah=0
        dw      d_check ;ah=1
        dw      none
        dw      init1    ;ah=3
        dw      status   ;ah=4
        dw      d_write ;ah=5
        dw      d_read
        dw      none
        dw      none
        dw      none
        dw      none
        dw      none
        dw      none
        dw      f_format;ah=0d
        dw      set_side
        dw      none
```

RAMディスクBIOSのコマンドテーブル

```
;
d_check:
        xor     ah,ah
        ret
;
init1:
        xor     ah,ah
        ret
;
status:
        xor     ah,ah
        ret
```

何もせずに戻る

```
;
f_format:
        cld
        mov     ax,disk_base
        mov     es,ax
        mov     di,0
        mov     al,0ffh
        mov     cx,1000h
        rep     stosb
;
        mov     di,0c00h
        mov     ax,0
        mov     cx,256
        rep     stosb
;
        mov     ax,0fefeh
        mov     es:[0d00h],ax
        mov     es:[0e00h],ax
        mov     es:[0f00h],ax
        xor     ah,ah
        ret
```

RAMディスクのフォーマット，ディレクトリ，FAT領域のイニシャライズを行う

```
;
set_side:
        xor     ah,ah
        ret
```

何もせずに戻る

```
;
d_write:
```

```
d_read:
        mov     cx,[bp+4]                ]
        cmp     cx,max_track_no+1        ] トラック番号のチェック
        jnb     error1                   ]
        mov     dx,[bp+6]                ]
        dec     dx                       ]
        cmp     dx,16                    ] セクタ番号のチェック
        jnb     error1                   ]
;
        mov     bx,[bp+2]                ]
        mov     ax,[bp+8]                ] DMAバウンダリのチェック
        add     ax,bx                    ]
        jc      error2                   ]
;
        xchg    ch,cl                    ]
        mov     ax,16                    ]
        mul     dx                       ] トラック，セクタ番号からメモリ上のアドレス
        add     ax,cx                    ] (セグメント)を得る
        add     ax,disk_base             ]
;
        push    ax                       ]
        mov     cx,[bp+2]                ]
        mov     al,[bp+1]                ] アドレスをPUSHしREAD,WRITEのルーチンへ
        and     al,0fh                   ] 分岐する
        cmp     al,5                     ]
        jz      write1                   ]
read1:
        pop     ds                       ]
        mov     si,0                     ]
        mov     es,[bp+10]               ] RAMディスクのメモリからデータを転送する
        mov     di,[bp+8]                ]
        rep     movsb                    ]
        jmp     short return1            ]
;
write1:
        pop     es                       ]
        mov     di,0                     ]
        mov     ds,[bp+10]               ] RAMディスクへデータを転送する
        mov     si,[bp+8]                ]
        rep     movsb                    ]
return1:
        xor     ah,ah                    ] 正常な終わり
        ret                              ]
;
error1:
        mov     ah,80h                   ]
        ret                              ]
error2:                                  ]
        mov     ah,20h                   ]
        ret                              ] エラーが発生した時の終了
;                                        ]   80H:エラー
none:                                    ]   20H:転送する領域がセグメント境界にかかっている
        popf            ;dummy           ]   40H:ECエラー
        mov     ah,40h                   ]
        jmp     exit1                    ]
;
main    ends
        ;
        end
```

8-3-4 RAMディスクの使用法と注意点

RAMディスクはリスト8-3-1をMASMでアセンブルし，BINファイルを作成して，BASICディスク上に"L8-3-1"というファイル名で格納したうえで，リスト8-3-2を実行することにより使用可能になります．ドライブ番号は，その時どのようなディスクが何台接続されているかによって異なります．

リスト 8-3-2 起動プログラム

```
100 ' ram-disk startup by m.f.
110 DEF SEG=&H1C00
120 BLOAD "L8-3-1"
130 A=0:CALL A
140 END
```

ところで，RAMディスクはディスクとは言っても電源を切ったり，リセットをすれば内容が失われてしまいます．また，現在は問題なく動作していますが，LIOとBASICとのインターフェイスについての情報が未公開なこともあり，メーカーの保障外の設定を行っているため将来の仕様変更に対応できるかどうか気がかりな点もあります．したがって，重要なファイルに関しては，必ずフロッピーディスクに書き込んだうえで，ワーク用として使用するように心がけてください．

8-4 簡単なインタープリタを作る

8-4-1 プログラムの仕様

ここでは，マシン語を使って簡単な逐時実行型言語（インタープリタ）を作成してみましょう．

インタープリタというとBASICなどの巨大な(?)プログラムを想像されるかもしれませんが，ここでは，マシン語による言語の作成テクニックの解説を目的とするために，紹介するのはFORTHと呼ばれる処理系をもとにした簡単な例で，そのまま実用になるものではありません．また，マシン語プログラムの内部構成を主に説明したいと思いますので，言語自体についてはあまり触れないことにしました（リスト8-4-1,8-4-2)．

このプログラム（リスト8-4-1）をアセンブルして得られたBINファイルを，BASICディスクに転送する際は，オフセットに&H4000を指定してください．起動にはリスト8-4-2を実行します．起動後は"=="のプロンプトが表示されて，入力待ちになります．この状態を実行モードと呼び，BASICのダイレクト・モードのように入力した命令は即実行されます．命令はスペースまたはリターンで区切られたワードで，第1文字目が数字（0～9）であれば，それは16進表現の数値として扱われます．

このモードで"："(コロン)が実行されると登録モードになり，次のワードを名前として"；"（セミコロン）が現われるまでの命令文を，あらたにひとつの命令として登録します．一度登録した命令は以後その名前で参照することができます．つまり新しい命令を追加，登録していくことが，このインタープリタでのプログラミングになるのです．

リスト 8-4-1 簡易インタープリタ

```
;       simple cli 2/10 by m.f.
;
null            equ     0000h       ――ディクショナリのリンクの終わりを示す
rstack          equ     8000h       ――リターン・スタック・ポインタの初期値
pstack          equ     0e000h      ――パラメータスタックの初期値
;
main    segment
        org     4000h               ――プログラムは60:4000Hからロードされる
        assume  cs:main,ds:main,es:main
start:
        mov     axt,ax              ┐
        mov     dst,ds              │ BASICへ戻るために保存する
        mov     bxt,bx              │
        mov     spt,sp              ┘
        mov     ax,cs               ┐
        mov     ds,ax               │ セグメント・レジスタは60Hにする
        mov     es,ax               ┘
        mov     lastins,offset basic ┐ ディクショナリの最終ワードの設定
        mov     opoint,offset enddic ┘
;
        mov     sp,pstack           ┐ リターンスタックとパラメータスタックの
        mov     bp,rstack           ┘ イニシャライズ
        mov     state,0             ┐
        mov     namflg,0ffh         │ 実行モードにしてtextiへ
        jmp     texti               ┘
outprm:
        cmp     state,0             ┐
        jz      er1                 │
        ret                         │
er1:    mov     al,'='              │ 実行モードであればプロンプトを出す
        call    out1                │
        call    out1                ┘
        ret
;
err1:
        mov     al,'?'              ┐
        call    out1                │ エラー時??を出力する
        call    out1                │ パラメータスタックを再設定する
        mov     sp,pstack           │
        jmp     texti               ┘
;
texti1: dw      $+2
texti2: dw      t2
texti:  call    outprm              ――プロンプト表示
texti3: call    getline             ――1行入力
t2:     mov     ax,pstack           ┐
        cmp     ax,sp               │ パラメータスタックをチェックする
        jc      err1                ┘
        call    getword             ――ワードを切り出してwbuffへ転送する
        jc      texti
        cmp     state,0             ┐ 登録モードならsetdicへ
        jnz     setdic              ┘
        call    sarhdic             ┐ ディクショナリを走査してそのワードがあれ
        jnc     exec                ┘ ば実行する
        call    num                 ┐ なければ数値として変換する
        jc      err1                ┘ 変換できなければエラー
        push    dx                  ――数値はパラメータスタックに積む
        mov     bx,offset texti2
```

```
        cmp     al,0ah
        jz      gl4
        cmp     al,20h
        jz      gl1
        cmp     al,0
        jz      gl4
gl2:    inc     di
        mov     [di+offset wbuff],al
        mov     al,[bx]
        cmp     al,0
        jz      gl3
        cmp     al,20h
        jz      gl3
        inc     bx
        jmp     gl2
gl3:    mov     tpoint,bx
        mov     bx,di
        mov     wbuff,bl
        inc     di
        mov     [di+offset wbuff],byte ptr 0
        clc
        ret
gl4:    stc
        ret
;
sarhdic:
        cld
        mov     bx,lastins
sl1:    mov     al,[bx]
        xor     ah,ah
        and     al,0fh
        test    [bx],byte ptr 80h
        jz      sl4
        cmp     al,wbuff
        jz      sl2
sl4:    add     bx,ax
        inc     bx
        mov     bx,[bx]
        cmp     bx,0
        jnz     sl1
        stc
        ret
sl2:    mov     cx,ax
        mov     si,bx
        inc     si
        mov     di,offset wbuff+1
sl3:    cmpsb
        jnz     sl4
        loop    sl3
        mov     di,bx
        add     ax,3
        add     bx,ax
        clc
        ret
;
num:
        mov     cl,4
        mov     ah,wbuff
        mov     si,offset wbuff+1
```

1行入力したデータを走査していき，ワードに切ってそれをwbuffに転送する．wbuffには文字数，ワード本体，デリミタ(0)が書かれる

ディクショナリを走査してwbuffの内容と一致するワードを探し，そのワードの先頭アドレスをdiに，実行アドレスをbxにいれる．見つからなければキャリーをセットする
走査はディクショナリの最後から行う

```
exec:   mov     si,offset texti1      ; ワードの実行アドレスへジャンプしてその処理を行う
        mov     di,offset next
        jmp     [bx]
;
setdic:                               ; ディクショナリへ新たにワードを追加する
        cld
        cmp     namflg,0              ; wbuffの内容をそのワードの名前とする
        jz      namset
        call    sarhdic               ; そのワードがあればse1へ
        jnc     se1
        call    num                   ; 数値への変換
        jc      err1
        mov     bx,offset lit1        ; 数値の登録。lit1へのポインタに続いて値が登録される
        call    se2
        mov     bx,dx
        call    se2
        jmp     t2
 se1:
        test    [di],byte ptr 40h     ; ワードの第6ビットがセットされていれば登録モードでもそれを実行する
        jnz     exec
        call    se2
        jmp     t2
 se2:   mov     di,cpoint             ; ディクショナリ部にそのワードの処理(ポインタ)を書き込んでいく
        mov     [di],bx
        inc     di
        inc     di
        mov     cpoint,di
        ret
 ;
 namset:                              ; ワードのヘッダ(属性,名前,リンクポインタ)を作成する
        mov     al,wbuff
        cmp     al,16
        jnb     err1
        or      al,80h
        mov     di,cpoint
        stosb
        mov     cl,wbuff
        mov     ch,0
        mov     si,offset wbuff+1
        rep     movsb
        mov     ax,lastins
        stosw
        mov     ax,offset entry
        stosw
        mov     cpoint,di
        mov     namflg,0ffh
        jmp     t2
 ;
 getline:                             ; キーボードから1行入力する データは202Hから格納される
        call    keyin2
        jcxz    hotj
        mov     tpoint,202h
        ret
 hotj:  pop     ax
        jmp     texti
 ;
 getword:mov    bx,tpoint
        mov     di,0
 gl1:   mov     al,[bx]
        inc     bx
```

```
        cmp     al,0ah
        jz      gl4
        cmp     al,20h
        jz      gl1
        cmp     al,0
        jz      gl4
gl2:    inc     di
        mov     [di+offset wbuff],al
        mov     al,[bx]
        cmp     al,0
        jz      gl3
        cmp     al,20h
        jz      gl3
        inc     bx
        jmp     gl2
gl3:    mov     tpoint,bx
        mov     bx,di
        mov     wbuff,bl
        inc     di
        mov     [di+offset wbuff],byte ptr 0
        clc
        ret
gl4:    stc
        ret
```

1行入力したデータを走査していき，ワードに切ってそれをwbuffに転送する．wbuffには文字数，ワード本体，デリミタ(0)が書かれる

```
;
sarhdic:
        cld
        mov     bx,lastins
sl1:    mov     al,[bx]
        xor     ah,ah
        and     al,0fh
        test    [bx],byte ptr 80h
        jz      sl4
        cmp     al,wbuff
        jz      sl2
sl4:    add     bx,ax
        inc     bx
        mov     bx,[bx]
        cmp     bx,0
        jnz     sl1
        stc
        ret
sl2:    mov     cx,ax
        mov     si,bx
        inc     si
        mov     di,offset wbuff+1
sl3:    cmpsb
        jnz     sl4
        loop    sl3
        mov     di,bx
        add     ax,3
        add     bx,ax
        clc
        ret
```

ディクショナリを走査してwbuffの内容と一致するワードを探し，そのワードの先頭アドレスをdiに，実行アドレスをbxにいれる．見つからなければキャリーをセットする
走査はディクショナリの最後から行う

```
;
num:
        mov     cl,4
        mov     ah,wbuff
        mov     si,offset wbuff+1
```

```
        lodsb
        cmp     al,'0'
        jb      ne1
        cmp     al,'9'+1
        jnb     ne1
        mov     dx,0
num1:   sub     al,30h
        cmp     al,9+1
        jb      num2
        and     al,11011111b        wbuffの内容を16進の数値としてバイナリ・
        sub     al,7                データに変換する.
        cmp     al,0ah              1文字目は数字でなければならない.
        jb      ne1                 4桁以上ある場合は最後の4桁が変換される.
        cmp     al,10h              負数などは考えていない
        jnb     ne1
num2:   shl     dx,cl
        add     dl,al
        lodsb
        cmp     al,0
        jnz     num1
        mov     numb,dx
        clc
        ret
ne1:    stc
        ret
;
axt     dw      ?
bxt     dw      ?                   レジスタの保存場所
dst     dw      ?
spt     dw      ?
numb    dw      ?
lastins dw      ?―――――――――― ディクショナリの最終ワードの先頭アドレス
tpoint  dw      ?―――――――――― キー入力バッファ内のポインタ
opoint  dw      ?―――――――――― ディクショナリの最終ワードの実行アドレス
cpoint  dw      ?―――――――――― 登録中のディクショナリ内のポインタ
namflg  db      ?―――――――――― 0ならwbuffを名前とする
state   db      ?―――――――――― 0なら実行モード
wbuff   db      257 dup(?)          キー入力バッファから切り出したワードの格納場所
;
entry:  dec     bp
        dec     bp
        mov     [bp],si
        mov     si,bx
        inc     si
        inc     si
next:   mov     bx,[si]             実行制御部
        inc     si
        inc     si
        jmp     [bx]
return  dw      $+2
        mov     si,[bp]
        inc     bp
        inc     bp
        jmp     di
;
val:    inc     bx
        inc     bx                  変数の処理ルーチン
        push    bx
        jmp     di
;
```

```
lit1    dw      $+2
        push    [si]
        inc     si          数値の処理ルーチン
        inc     si
        jmp     di
;
jumpz1  dw      $+2
        pop     ax
        or      ax,ax
        jnz     jumpz2
jump2:  mov     ax,[si]
        add     si,ax
        jmp     di          分岐の処理ルーチン
jumpz2: inc     si
        inc     si
        jmp     di
;
jump1   dw      jump2
;
;dictionary
;
at      db      81h
        db      '@'
        dw      null
at1     dw      $+2         @の処理
        pop     bx
        push    [bx]
        jmp     di
;
store   db      81h
        db      '!'
        dw      at
store1  dw      $+2         !の処理
        pop     bx
        pop     ax
        mov     [bx],ax
        jmp     di
;
tadd    db      81h
        db      '+'
        dw      store
tadd1   dw      $+2
        pop     bx          +の処理
        pop     ax
        add     ax,bx
        push    ax
        jmp     di
;
tsub    db      81h
        db      '-'
        dw      tadd
tsub1   dw      $+2
        pop     bx          -の処理
        pop     ax
        sub     ax,bx
        push    ax
        jmp     di
tand    db      83h
        db      'and'
        dw      tsub
```

```
tand1   dw      $+2                 and の処理
        pop     bx
        pop     ax
        and     ax,bx
        push    ax
        jmp     di
;
tor     db      82h
        db      'or'
        dw      tand
tor1    dw      $+2                 or の処理
        pop     bx
        pop     ax
        or      ax,bx
        push    ax
        jmp     di
;
txor    db      83h
        db      'xor'
        dw      tor
txor1   dw      $+2
        pop     bx                  xor の処理
        pop     ax
        xor     ax,bx
        push    ax
        jmp     di
;
tnot    db      83h
        db      'not'
        dw      txor
tnot1   dw      $+2
        pop     ax                  not の処理
        not     ax
        push    ax
        jmp     di
;
teq     db      81h
        db      '='
        dw      tnot
teq1    dw      $+2
        pop     bx                  =の処理
        pop     ax
        cmp     ax,bx
        je      true
        jmp     false
;
tge     db      82h
        db      '>='
        dw      teq
tge1    dw      $+2
        pop     bx                  >=の処理
        pop     ax
        cmp     ax,bx
        jae     true
        jmp     false
;
tle     db      82h
        db      '<='
        dw      tge
tle1    dw      $+2
        pop     bx                  <=の処理
```

```
        pop     ax
        cmp     ax,bx
        jna     true
        jmp     false
;
tg      db      81h
        db      '>'
        dw      tle
tg1     dw      $+2
        pop     bx                  >の処理
        pop     ax
        cmp     ax,bx
        ja      true
        jmp     false
;
tl      db      81h
        db      '<'
        dw      tg
tl1     dw      $+2
        pop     bx
        pop     ax
        cmp     ax,bx
        jnae    true                <の処理
false:  mov     ax,0
        push    ax
        jmp     di
true:   mov     ax,0ffffh
        push    ax
        jmp     di
;
seg0    db      83h
        db      'seg'
        dw      tl
seg1    dw      val
        dw      ?
v1      db      82h
        db      'v1'
        dw      seg0
v11     dw      val
        dw      ?
v2      db      82h
        db      'v2'
        dw      v1                  変数の処理とデータ領域
v21     dw      val
        dw      ?
v3      db      82h
        db      'v3'
        dw      v2
v31     dw      val
        dw      ?
v4      db      82h
        db      'v4'
        dw      v3
v41     dw      val
        dw      ?
;
sload   db      82h
        db      's@'
        dw      v4
sload1  dw      $+2
        pop     bx
```

```
        pop     ax
        push    ds
        mov     ds,ax
        mov     ax,[bx]
        pop     ds
        push    ax
        jmp     di
```

S@ の処理
パラメータ・スタックの2ndの値をセグメント値とし、topの値をオフセットとしてそのアドレスのデータをパラメータスタックにPUSH

```
;
dup0    db      83h
        db      'dup
        dw      sload
dup1    dw      $+2
        pop     ax
        push    ax
        push    ax
        jmp     di
```

dup の処理
パラメータスタックのtop をコピーする

```
;
swap0   db      84h
        db      'swap'
        dw      dup0
swap1   dw      $+2
        pop     ax
        pop     bx
        push    ax
        push    bx
        jmp     di
```

swap の処理
パラメータ・スタックの2ndとtop を入れ換える

```
;
over0   db      84h
        db      'over'
        dw      swap0
over1   dw      $+2
        pop     bx
        pop     ax
        push    ax
        push    bx
        push    ax
        jmp     di
```

over の処理
パラメータ・スタックの2ndをコピーする

```
;
i       db      81h
        db      'i'
        dw      over0
i1:     dw      $+2
        push    [bp]
        jmp     di
```

i の処理
リターン・スタックのtop をパラメータ・スタックにPUSH

```
;
call0   db      05h
        db      'call0'
        dw      i
call01  dw      $+2
        pop     ax
        or      ax,ax
        jnz     12
        inc     si
        inc     si
        dec     bp
        dec     bp
        mov     [bp],si
        mov     si,[si]
12:     inc     si
        inc     si
        jmp     di
```

先頭アドレスのMSBがクリアされているので、sarhdlc で無視される

```
;
do      db      0c2h
        db      'do'
        dw      call0
do1     dw      $+2
        cmp     state,0
        jnz     do2
        jmp     di
do2:    mov     bx,cpoint
        mov     ax,offset doe
        mov     [bx],ax
        inc     bx
        inc     bx
        push    bx
        mov     cpoint,bx
        jmp     di
```

登録モード時のdoの処理
パラメータ・スタックに次のアドレスをPUSHする

```
doe     dw      $+2
        pop     ax
        pop     dx
        mov     [bp-2],dx
        mov     [bp-4],ax
        sub     bp,4
        jmp     di
```

実行時のdoの処理
パラメータ・スタックの最大値とカウンタ値(初期値)をリターン・スタックへコピーする.

```
;
loop    db      0c4h
        db      'loop'
        dw      do
loop1   dw      $+2
        cmp     state,0
        jnz     lo1
        jmp     di
lo1:    mov     bx,cpoint
        mov     ax,offset loope
        mov     [bx],ax
        inc     bx
        inc     bx
        pop     ax
        sub     ax,bx
        mov     [bx],ax
        inc     bx
        inc     bx
        mov     cpoint,bx
        jmp     di
```

登録モード時のloopの処理
doの処理の時PUSHしたアドレスへのオフセットを計算してセットする

```
loope   dw      $+2
        mov     ax,[bp]
        inc     bp
        inc     bp
        inc     ax
        cmp     ax,[bp]
        jnb     lo2
        dec     bp
        dec     bp
        mov     [bp],ax
        mov     ax,[si]
        add     si,ax
        jmp     di
lo2:    mov     ax,[bp]
        inc     bp
        inc     bp
        inc     si
        inc     si
```

実行時のloop処理
リターン・スタックの最大値とインクリメントしたカウンタ値を比較し，最大値以上になればループを抜ける

```
        jmp     di
;
if00    db      0c2h
        db      'if'
        dw      loop
if01    dw      $+2
        cmp     state,0
        jnz     if02
        jmp     di
if02:   mov     bx,cpoint
        mov     ax,offset jumpz1
        mov     [bx],ax
        inc     bx
        inc     bx
        push    bx
        inc     bx
        inc     bx
        mov     cpoint,bx
        jmp     di
```

登録モード時のifの処理
jumpzへのポインタをセットし，オフセットのはいるアドレスをPUSHする．
パラメータ・スタックのtopがfalse(0)のとき分岐させるため

```
;
else00  db      0c4h
        db      'else'
        dw      if00
else01  dw      $+2
        cmp     state,0
        jnz     else02
        jmp     di
else02: mov     bx,cpoint
        mov     ax,offset jump1
        mov     [bx],ax
        inc     bx
        inc     bx
        mov     cx,bx
        inc     bx
        inc     bx
        mov     ax,bx
        mov     dx,bx
        pop     bx
        sub     dx,bx
        mov     [bx],dx
        push    cx
        mov     cpoint,ax
        jmp     di
```

登録モード時のelseの処理
ifのjumpzのオフセットを計算してifの処理を完成させる．またtrueの時の処理の終わりからendifへのjump命令をセットし，このjump命令のオフセットのはいるアドレスをPUSHする

```
;
endif0  db      0c5h
        db      'endif'
        dw      else00
endif1  dw      $+2
        cmp     state,0
        jnz     endif2
        jmp     di
endif2: mov     dx,cpoint
        pop     bx
        sub     dx,bx
        mov     [bx],dx
        jmp     di
```

登録モード時のendifの処理
パラメータ・スタックのtopの値と現在のアドレスからオフセット値を求め，その値をセットする

```
keyin   db      85h
        db      'keyin'
        dw      endif0
keyin1  dw      $+2
```

keyinの処理
キーボードから1行入力を行う

```
        call    keyin2
        push    cx
        jmp     di
keyin2: push    di
        mov     di,82
        jmp     out2
;
outb    db      81h
        db      '.'
        dw      keyin
outb1   dw      $+2
        pop     bx
        jmp     outb2
```

・の処理
パラメータ・スタックのtopの下位バイトを16進2桁で出力する

```
;
outw    db      82h
        db      '..'
        dw      outb
outw1   dw      $+2
        pop     bx
        mov     ah,bh
        call    hl1
outb2:  mov     ah,bl
        call    hl1
        jmp     di
hl1:    mov     cl,4
        mov     al,ah
        shr     al,cl
        call    hl2
        mov     al,ah
        and     al,0fh
        call    hl2
        ret
hl2:    add     al,30h
        cmp     al,3ah
        jc      out1
        add     al,7
out1:   push    di
        mov     di,61
out2:   push    ax
        push    bx
        push    si
        int     0c4h
        pop     si
        pop     bx
        pop     ax
        pop     di
        ret
```

・・の処理
パラメータ・スタックのtopを16進4桁で出力する

```
;
cr      db      82h
        db      'cr'
        dw      outw
cr1     dw      $+2
        mov     al,0dh
        call    out1
        mov     al,0ah
        call    out1
        jmp     di
```

crの処理
cr, lfのコードを出力する

```
;
space   db      85h
        db      'space'
        dw      cr
```

spaceの処理
スペースを出力する

```
space1  dw      $+2
        mov     al,20h
        call    out1
        jmp     di
;
outp0   db      84h
        db      'outp'
        dw      space
outp1   dw      $+2
        pop     ax
        mov     ah,11h
        push    si
        push    bp
        int     1ah
        pop     bp
        pop     si
        jmp     di
```

outpの処理
プリンタへパラメータ・スタックのtopの下位バイトを出力する

```
;
colon   db      81h
        db      ':'
        dw      outp0
colon1  dw      $+2
        cmp     state,0
        jnz     colon2
        mov     state,0ffh
        mov     namflg,0
        mov     ax,opoint
        mov     cpoint,ax
colon2: jmp     di
```

:の処理
登録モードにする
次にくるワードを名前とするようnamflgを0にする

```
secol   db      0c1h
        db      ';'
        dw      colon
secol1  dw      $+2
        cmp     state,0
        jz      sec1
        mov     bx,cpoint
        mov     ax,offset return
        mov     [bx],ax
        inc     bx
        inc     bx
        mov     state,0
        mov     ax,opoint
        mov     lastins,ax
        mov     opoint,bx
sec1:   jmp     di
```

;の処理
lastinsとopointを更新してワードの登録を完了する
実行モードにする

```
basic   db      85h
        db      'basic'
        dw      secol
basic1  dw      $+2
        mov     ax,axt
        mov     bx,bxt
        mov     ds,dst
        mov     sp,spt
        iret
```

basicの処理
basicのコマンドレベルへ戻る

```
        dw      0
        dw      0
enddic:
;
main    ends
        end
```

リスト 8-4-2　起動プログラム

```
100 '    18-4-2
110 DEF SEG=&H60
120 BLOAD 'L8-4-1',&H4000
130 COLD=&H4000: CALL  COLD
140 END
```

命令文の構成は，逆ポーランド記法（post fix 記法）に従います．詳しくは FORTH 等の文献を参照していただくとして，ここでは実行例をいくつかリスト **8-4-3** に示しておきます．

リスト 8-4-3　実行例

```
==100 ―――――――――――――――――――――――――――――― 100Hをパラメータ・スタックにPUSHする
==.. ――――――――――――――――――――――――――――――― パラメータ・スタックのtopを表示する
0100 ――――――――――――――――――――――――――――――― 100Hが表示された
==100 200 300 .. .. ..                    ] 3つの値をPUSHしてそれを表示する
03000200100==                             ] スタックのためデータが逆になっている
==100 200 300 .. space .. space .. cr     ] データをスペースで区切って出す
0300 0200 0100                            ]
==100 200 + .. cr                         ] 100+200の結果を表示し，改行する
0300                                      ]
==100 200 300 + + .. cr                   ] 100+200+300の結果を表示し，改行する
0600                                      ]
==100 200 + 300 + .. cr ―――――――――――――――― このような表現も可能
0600
==5555 not .. cr                          ] 5555Hを反転して表示，改行する
AAAA                                      ]
==5555 00ff and .. cr                     ] 5555Hと00FFHのANDをとり表示，改行する
0055                                      ]
==
404F??==..                                ] パラメータ・スタックに値が積まれていないのに表示(ポップ)
404F??==..                                ] しようとしたためのエラー．登録時はチェックがないので注意
404F??==
==100 v1 ! ――――――――――――――――――――――――――― 変数v1に100Hを代入する
==v1 @ .. cr                              ] v1のデータをスタックにPUSHしたうえで，
0100                                      ] POPして表示，改行する
==200 v2 ! ――――――――――――――――――――――――――― 変数v2に200Hを代入する
==v1 @ v2 @ + .. cr                       ] v1+v2の結果を表示，改行する
0300                                      ]
==
==100 200 swap .. space .. cr             ] パラメータ・スタックのtopと2ndを入れ換えて表示，改行する
0100 0200                                 ]
==100 200 dup .. space .. space .. cr     ] 200Hをコピー(さらにスタックに積む)して表示，改行する
0200 0200 0100                            ]
==
==: test1 100 200 .. space .. space cr ;― test1という名前で，crまでの処理を定義する
==test1                                   ] test1を実行する
0200 0100                                 ]
==: test2 + + .. cr ; ―――――――――――――――――― 3つのデータを加えるtest2を定義する
==100 200 300 test2                       ] データを先にスタックに積んでおいてtest2を実行する
0600                                      ] すなわちパラメータをtest2に与えることになる
==: test3 0a 0 do 0f0 . space loop cr ;   ] 10回F0Hを表示し改行する
==test3                                   ] 0AHがループの最終値で，0が初期値
F0 F0 F0 F0 F0 F0 F0 F0 F0 F0             ]
==: test4 do 0f0 . space loop cr ; ――――― do～loopの間にループ回数をパラメータとして考える
==5 0 test4                               ] 5回表示の例
F0 F0 F0 F0 F0                            ]
==5 5 test4                               ] 1回は必ず実行される
F0                                        ]
==
==: test5 0a 0 do dup . space loop cr ;   ] 表示データを与えてそれを10回表示し，改行する
==0f5 test5                               ]
F5 F5 F5 F5 F5 F5 F5 F5 F5 F5             ]
```

※"."，".."による表示により，スタックの内容はPOPされてなくなる点に注意

```
==
==: test6 10 0 do i . space loop cr ;                    ループのカウンタ値を取り出して
==test6                                                  それを表示する
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
==
==: test7 4 0 do test6 loop ; ---------------------------- test6を4回実行するtest7を定義
==test7
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
==: test8 8 1 do i 0 do i . space loop cr loop ; --- 行数と同じ数だけカウンタ値を表
==test8                                              示する例．doのネストが行われて
00                                                   いる
00 01
00 01 02
00 01 02 03
00 01 02 03 04
00 01 02 03 04 05
00 01 02 03 04 05 06
==                                                   パラメータ・スタックのtopが0なら
==: test9 if 100 .. else 200 .. endif cr ; ------    else以下の処理を，そうでなければ
==0ffff test9                                        if～elseの間の処理を行う
0100                                                 true(＝0)の場合
==0 test9
0200                                                 falseの場合
==
==: test10 if 300 .. endif cr ; ------------------- elseはなくてもよい
==0ffff test10
0300                                                 trueの場合(≠0)
==0 test10
                                                     false(＝0)の場合，crだけ実行される
==
==: test11 if if 100 .. else 200 .. endif else 300 .. endif cr ;
==0 0 test11
0300
==0 1 test11
0200
==1 0 test11                                         ifのネストの例
0300
==1 1 test11
0100
==
==: p11 0a0 0 do seg @ v1 @ s@ outp v1 @ 1 + v1 ! loop 0a outp ;   テキストV-RAM
==: pv 0a000 seg ! 0 v1 ! 18 1 do p11 loop ;                       の内容をプリンタ
==                                                                 に出力する．s@は，
                                                                   セグメントを指定
                                                                   できるロード命令
=: dump1 v1 @ .. space 10 0 do v1 @ dup @ . 1 + v1 ! space loop cr ;   与えたアド
==4000 v1 ! dump1                                                      レスから16
4000 A3 A8 41 8C 1E AC 41 89 1E AA 41 89 26 AE 41 8C                   バイトのダ
==: dump v1 ! 8  0 do dump1 loop ;  与えたアドレスから128バイトのダンプを行う   ンプを行う
==4000 dump
4000 A3 A8 41 8C 1E AC 41 89 1E AA 41 89 26 AE 41 8C
4010 C8 8E D8 8E C0 C7 06 B2 41 13 46 C7 06 B6 41 31
4020 46 BC 00 E0 BD 00 80 C6 06 BB 41 00 90 C6 06 BA
4030 41 FF 90 EB 24 90 80 3E BB 41 00 74 01 C3 B0 3D
4040 E8 42 05 E8 3F 05 C3 B0 3F E8 39 05 E8 36 05 BC
4050 00 E0 EB 05 90 57 40 5F 40 E8 DA FF E8 8D 00 B8
4060 00 E0 3B C4 72 E1 E8 93 00 72 EE 80 3E BB 41 00
4070 75 16 E8 C3 00 73 09 E8 F6 00 72 CB 52 BB 57 40
==
==:                          登録モードではリターンのみでは入力終了とはならないので，
dummy ;                      続けて入力することが可能
==: test6 print ;            未登録の命令を入力すると“??”が表示され，“;”の入力で実行
??;                          モードに戻る．ただし“:”や“;”をあらたに命令として登録して
==                           しまうと，以後登録ができなくなるので注意
==basic -------------------- BASICへの復帰命令
Ok
```

8-4-2 プログラムの内部構造

このプログラムは，実行制御部，テキスト処理部，ディクショナリ部から成っており，ワークエリアとしてリターン・スタックとパラメータ・スタックを用います．これらは図8-4-1のような領域に配置されます．

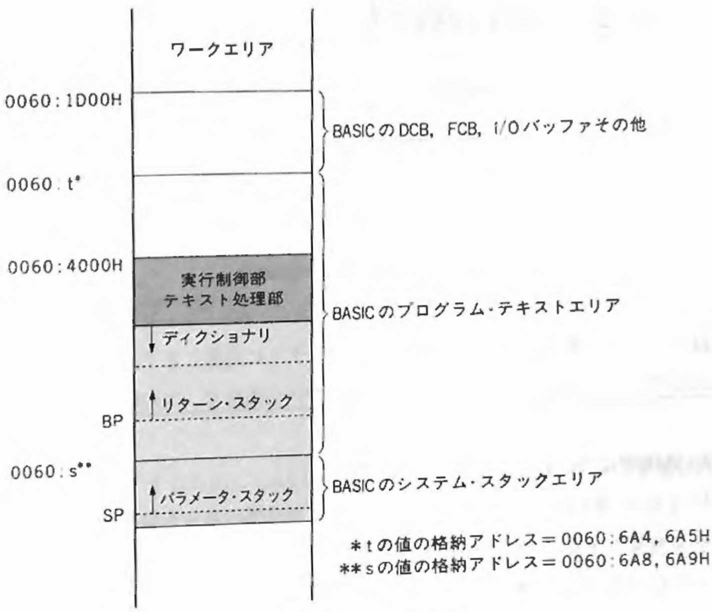


図8-4-1 メモリマップ

ここは本来BASICのプログラムが中間言語の形式で置かれる，テキスト領域ですが，ROM内ルーチンを使ううえで都合が良いのでここに置くことにしました．

ディクショナリ部は，命令の名前と処理が登録されている部分で，このインタープリタが実行するプログラムが書かれている領域と言っても良いでしょう．ディクショナリの構造を図8-4-2に示します．

各命令はリンク・ポインタで高番地から低番地（新しい登録命令から古い登録命令）に向かって結ばれており，この方向で検索されます．

パラメータ・スタックとは文字どおりパラメータをやり取りする領域で，処理するデータや結果は，ここを介して命令に渡されます．これは，SPによって実現される8086の通常のスタックを用いています．

リターン・スタックは，この処理系におけるサブルーチン・コールの戻り番地が格納される領域で，BPをポインタとしてスタックを実現しています．

テキスト処理部は，キーボードから命令の入力を行い，実行モード時はその命令を制御部で実

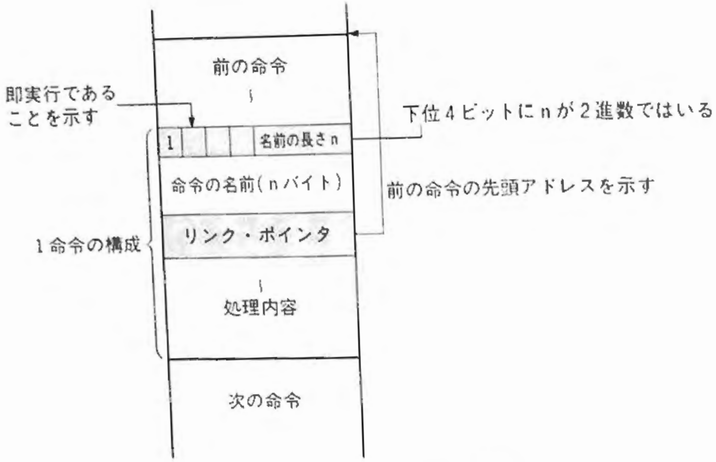


図8-4-2　ディクショナリの構造

行させ，登録モードでは新たにその命令をディクショナリに追加します．

実行制御部はディクショナリにある命令を実行していく部分で，次の節で詳しく説明します．

8-4-3　実行制御部について

インタープリタは，BASICでおなじみのように，その言語の命令を取り出して対応するマシン語処理ルーチンを実行する，という動作を行います．リスト8-4-4はマシン語プログラム例ですが，このようにCALL命令が羅列している場合，階層的に見れば，一種のインタープリタがあるように考えられます．たとえばMIZU, TOKEI, MOTOR以下のルーチンを不可視であるとすれば，ARAIという命令をそのインタープリタが取り出して実行すると，MIZU以下のルーチンが見えない部分で呼び出されることになります．

リスト 8-4-4　構成例1

```
;           L8-4-4
                :
SHIGOTO:
            CALL    SENTAKU
            CALL    SOUJI
            CALL    KAIMONO
                :
                :
            END
;
SENTAKU:
            CALL    ARAI
            CALL    SUSUGI
            CALL    KANSO
            RET
;
```

```
ARAI:   CALL    MIZU
        CALL    TOKEI
        CALL    MOTOR
        MOV     ...
          :
          :
        RET
;
SUSUGI:
          :
          :
        RET
;
KANSO:
          :
          :
        RET
;
MIZU:
          :
        RET
;
          :
          :
```

ここで，わかりやすくするためにCALL命令のオペコード部をなくして，処理名（実際はアドレス）の羅列を実行していくように書き換えると，リスト8-4-5のようになります．この時SIは，DWで設定した命令文を指す命令ポインタの役目を持っています．オペコード部がなくなった分，長いプログラムではメモリの節約にもなるはずです．

リスト 8-4-5　構成例2

```
;          L8-4-5
;
                :
SENTAKU:
           MOV     SI,OFFSET SHORI ———— SIに実行していく処理ルーチンのアドレスを指すポインタを入れる
           JMP     NEXT
SHORI      DW      ARAI       ┐
           DW      SUSUGI     │ 処理ルーチンのアドレスの羅列
           DW      KANSO      ┘
             :
             :
;----------------------------
ARAI:      CALL    MIZU       ┐
           CALL    TOKEI      │
           CALL    MOTOR      │
           MOV     ...        │
             :                │ それぞれの処理ルーチン
             :                │ この中ではポインタであるSIを破壊できない
           JMP     NEXT       │ 処理の終わりはRETでなくNEXTへの分岐になる
;                             ┘
```

```
SUSUGI:
                 :
                 :
        JMP      NEXT
;
KANSO:
                 :
                 :
        JMP      NEXT
;
;-----------------------------
NEXT:   MOV      BX,[SI]
        INC      SI         ; CALL命令のかわりにこのルーチンが次々と
        INC      SI         ; ルーチンを呼び出すことになる
        JMP      BX
;-----------------------------
```

このインタープリタでは，ARAI, SUSUGI, KANSO が命令の最小単位で，ARAI の処理ルーチン（CALL：MIZU……）などが，BASIC での ROM 内の処理ルーチンに対応することになります．さらに SENTAKU が，ARAI, SUSUGI, KANSO の命令で構成されたこのインタープリタ言語でのサブルーチンを表します．

次に考えることは，このインタープリタ・レベルでの階層化です．SENTAKU は，SHIGOTO のひとつのサブルーチンにすぎません．SENTAKU を終えると次の SOUJI へ処理を移さなければならないため，命令ポインタを保存する必要があります．この処理を行ったのがリスト 8-4-6 です．SHIGOTO ルーチンでの命令ポインタは，SENTAKU ルーチンで BP によって実現されるリターン・スタックへ格納され，終わったら RETURN によって戻されます．

リスト 8-4-6 階層化

```
;         L8-4-6
;
SHIGOTO:
          MOV       SI,OFFSET SHORI1   ――ポインタの設定
          JMP       NEXT
SHORI1    DW        SENTAKU
          DW        SOUJI
                    :
                    :
          END
;-----------------------------
;
SENTAKU:DEC         BP
          DEC       BP                 ］リターンスタックへのSIの退避(PUSH)
          MOV       [BP],SI
          MOV       SI,OFFSET SHORI    ――あらたにSIを設定する
          JMP       NEXT
SHORI     DW        ARAI
          DW        SUSUGI
          DW        KANSO
          DW        RETURN             ――これによりSENTAKU処理を終わり，SHIGOTO
                                         処理内のSOUJIルーチンを次に実行する
```

```
;
;------------------------------
;
ARAI:                              ┐
            :                      │
            :                      │
         JMP      NEXT             │
;                                  │ 最終的にはRETURNではなく，NEXTへ分岐する
SUSUGI:     :                      │ ルーチンに行きつく
         JMP      NEXT             │
KANSO:      :                      │
         JMP      NEXT             │
;                                  ┘
;------------------------------
;
NEXT:    MOV      BX,[SI]          ┐
         INC      SI               │ ポインタSIの指すアドレス中のデータを処理ルーチン
         INC      SI               │ のアドレスとしてそこに分岐する
         JMP      BX               ┘
;
RETURN:  MOV      SI,[BP]          ┐
         INC      BP               │ リターンスタックからSIを取り出し(POP)，NEXTへ分岐
         INC      BP               │
         JMP      NEXT             ┘
;------------------------------
```

これで，インタープリタ，命令レベルのサブルーチン・コールができるようになったわけですが，このままでは，コールするたびに各ルーチンの頭へリターン・スタックへの退避部分がくっつくことになります．そこでこれを共通化するようにしたのが，リスト 8-4-7 です．これはリスト 8-4-1 で使われているものと同じ方法になっています．

このリストで，SENTAKU のように ENTRY と RETURN で囲まれたアドレスの羅列はマシン（CPU）に依存しない部分です．これらのアドレスは，それぞれの処理ルーチンのアドレスを指し示すわけですが，アドレスだけでプログラムは成立しませんから，最終的には ARAI のようなマシン語ルーチンを示すことになります．この部分は，ENTRY の代わりに実際のマシン語を指すポインタがおかれ，最後は NEXT へ分岐して終わります．

この実行制御部は，FORTH の内部インタープリタと呼ばれる部分と同様な処理を行っています．

さて，ここまでの例では，各処理ルーチン間のパラメータのやり取りは行っていません．実際にはこのような例はまれで，ほとんどの処理では入力パラメータと出力パラメータが存在します．そこでリターン・スタックとは別にパラメータ・スタックを作り，これを介してデータのやり取りをします．スタックを使うことにより，メモリのロケーション等を考慮する必要がなくなりますから，これもマシン独立性を高める一因になります（リスト 8-4-7）．

リスト 8-4-7 パラメータの入出力 1

```
;          L8-4-7
;
           MOV      SI,OFFSET SHORI2   ] SIに最初の処理のアドレスをいれてNEXTへジャンプ
           JMP      NEXT               ]
SHORI2     DW       SHIGOTO            ] メインとなるルーチン
                :                      ]
           END
;
                :
SHIGOTO:
           DW       ENTRY              ] SHIGOTOルーチンの処理
           DW       SENTAKU            ]   アドレスの羅列にはENTRY,RETURNがその前後につ
           DW       SOUJI              ]   き，ポインタの制御と保存を行う
           DW       KAIMONO            ]
                :                      ]
           DW       RETURN             ]
;
SENTAKU:
           DW       ENTRY              ] SENTAKUルーチンの処理
           DW       ARAI               ]
           DW       SUSUGI             ]
           DW       KANSO              ]
           DW       RETURN             ]
;
ARAI:
           DW       $+2                ] 最終的に行きつくマシン語ルーチンでは，そのマシン語本体の
           CALL     MIZU               ] アドレスを示すポインタ(DW$+2)とNEXTへの分岐でくく
           CALL     TOKEI              ] る
           CALL     MOTOR              ] マシン語ルーチン内ではSIを保存しなければならない
           MOV      ...                ]
                :                      ]
                :                      ]
           JMP      NEXT               ]
;
MIZU:                                  ] マシン語サブルーチン
                :                      ]
           RET                         ]
;                                      ]
TOKEI:                                 ]
                :                      ]
           RET                         ]
;                                      ]
MOTOR:                                 ]
                :                      ]
           RET                         ]
;
;-----------------------------
;
NEXT:      MOV      BX,[SI]            ] SIの示すアドレスのデータが示す処理先アドレスへ分岐する
           INC      SI                 ]
           INC      SI                 ]
           JMP      [BX]               ]
;
ENTRY:     DEC      BP                 ] リターンスタックへポインタSIを保存
           DEC      BP                 ]
           MOV      [BP],SI            ]
           MOV      SI,BX
           INC      SI
```

```
         INC      SI
         JMP      NEXT  ──── ENTRYの次に書かれた処理ルーチンへ分岐する
;
RETURN:  DW       $+2
         MOV      SI,[BP]
         INC      BP          リターンスタックから，ポインタをSIに戻し,NEXTへ分岐
         INC      BP          する
         JMP      NEXT
;
;------------------------------
```

条件判断をして処理の流れを変える例として，パラメータ・スタックのトップの値がゼロであれば，命令ポインタをディスプレースメント分進めて分岐する命令，JUMPZ をリスト 8-4-8 に示します．

リスト 8-4-8　パラメータの入出力 2

```
;        L8-4-8
;
JUMPZ:   DW       $+2          JUMPZ処理本体
         POP      AX
         OR       AX,AX        パラメータ・スタックのトップの値を調べ0 ならば
         JZ       LZ1          LZ1 へ分岐する
         INC      SI
         INC      SI           そうでなければディスプレースメントの次にある処
         JMP      NEXT         理ルーチンを実行する
LZ1:     MOV      AX,[SI]
         ADD      SI,AX        ディスプレースメントを加えたポインタの指す処理
         JMP      NEXT         ルーチンを実行する
;------------------------------
;
                  :
                  :
                                                JUMPZの使用例
         DW       JUMPZ
D1       DW       OFFSET D2-OFFSET D1 ──── D2の処理へのディスプレースメント
         DW       NOT_ZERO ─────────────── 0 でなかった時の処理
                  :
D2       DW       ZERO ─────────────────── 0 の時の処理
                  :
```

ただし，分岐先までの距離を設定しなければならないので，実行モードでは使うことができません．また，このままではマシン語命令とあまり変わらないため，これを用いる上位の命令として IF 文やループ制御文が定義されます．

8-4-4 変数について

パラメータ・スタックは一時的なデータの保存には便利ですが，どの命令からも好きな時にデータを参照したい場合には不向きです．そこで，データに対して名前でアクセスできる，いわゆる変数が必要になります．

このインタープリタでは，変数も命令と同じように名前とその処理をディクショナリ中に登録し，値のはいる領域もそれに続けて確保されます．

しかし，リスト 8-4-1 では v1～v4 までをあらかじめ宣言してあり，新たに登録を行う処理部は省いてあるので，より以上の変数を使用したい方は，変数登録ルーチンを自作する必要があります．

変数の名前が入力されると，その変数の値のあるアドレスがパラメータ・スタックに積まれますので，その後そのアドレスを読み出す命令（@）や代入する命令（！）などで値をアクセスします．リスト 8-4-9 に変数の例を示します．なお，これと同様な考え方で定数や，配列を実現することが可能です．

リスト 8-4-9 変数の例

```
;        L8-4-9
;
                 :
VAL:     INC     BX         ┐
         INC     BX         │変数の値のはいるアドレスをパラメータスタックへPUSHする
         PUSH    BX         │
         JMP     NEXT       ┘
;
LOAD:    DW      $+2        ┐
         POP     BX         │パラメータスタックのトップの値をアドレスとしてその内容をスタ
         PUSH    [BX]       │ックに積む
         JMP     NEXT       ┘
;
STORE:   DW      $+2        ┐
         POP     BX         │パラメータスタックのトップをアドレスとし，そのアドレスへ（ト
         POP     AX         │ップ－2）のデータを書く
         MOV     [BX],AX    │
         JMP     NEXT       ┘
;----------------------------
;
V1       DW      VAL        ┐
         DW      ?          │V1，V2という2つの2バイト変数が宣言されている
V2       DW      VAL        │
         DW      ?          ┘
                 :
;----------------------------
;
                 :
                 :          変数使用例(V1のV2へのコピー)
         DW      V1         ┐V1の値がはいるアドレスをパラメータスタックに積む
         DW      LOAD       │そのアドレスの内容をパラメータスタックに積む
         DW      V2         │V2の値がはいるアドレスをパラメータスタックに積む
         DW      STORE      ┘V2へV1のデータを代入する
                 :
                 :
;
```

8-4-5　FORTHとの相違

このプログラムはFORTHを参考にしていますが，その洗練された構造に比べ，かなり雑な作りになっています．たとえばFORTHでは，できるだけFORTHレベルの命令で自身が記述されており，マシン独立性が非常に高くなっています．

またFORTHでは実行モードと登録モード（コンパイル・モード）で異なる処理を行う場合でもディクショナリ中のエントリは同じで，そこに制御が移ってからモードによりそれに合った処理が行われます．しかしこのプログラムでは，登録モードで行う処理をテキスト処理部でまとめて行っているため，拡張性が制限されています．

APPENDIX 1

MS-DOS・BASIC トランスファー・プログラム

リスト A-1-1, 2は，MS-DOSのルート・ディレクトリにあるファイルをDISK-BASICフォーマットのファイルに転送する時に使います．詳しい使用法については6-2-2を参照していただくとして，ここではプログラムの構成を説明します．

ファイルの転送でネックになるのは，両者の物理的フォーマットの違いです．DISK-BASICのコマンドでは，MS-DOSのフォーマットである，512バイト／セクタ，1024バイト／セクタのディスクの内容を読み出すことができません．そこでMS-DOSのファイルを読む場合のセクタ・リードには，マシン語ルーチンからBIOSのリード・コマンドを使っています．BIOSはDISK-BASICのものですが，8インチ，5インチ2DDのドライブに関しては，パラメータを変えることにより，それぞれ1024バイト／セクタ，512バイト／セクタのフォーマットを読むことができるのです．ただ残念なことに，5インチ2Dのドライブに対しては256バイト／セクタのフォーマットしか扱えないため，転送できるMS-DOSディスクに，特に制限があります．

リストのマシン語ルーチンは，CALL文により，8個のパラメータを渡しています．その意味と，とり得る値を図A-1-1に示します．BIOSコマンドについては「PC-9801システム解析(下)」をご覧ください．

マシン語ルーチンで読まれたプログラムは，B0000H～BFFFFHのV-RAM領域に格納され，BSAVE命令でDISK-BASICのディスクに書き込まれます．このプログラムは，マシン語データのみの転送を考えましたので，BSAVE文で簡単に済ませていますが，シーケンシャル・ファイルとして書き込むようにすれば，アスキーデータの転送も行うことができます．

MS-DOSのディスク・フォーマットのうち，8インチは両面倍密倍(FAT ID：FEH)，5インチは両面倍密(FAT ID：F9H，FBH，FDH，FFH)，5インチ両面倍密トラック，5インチ両面の高密度がサポートされます．しかし5インチ2Dのドライブ（80S31等）は，MS-DOSのシステム・ディスクのファイルのみ転送可能です．

このプログラムのほとんどは，MS-DOSのファイルを読んでV-RAMに展開するという動作に費やされています．その主なサブルーチンについて図A-1-2にまとめておきます．

エラーの処理は特別には行っておらず，エラーが生じたときは，エラー・メッセージを表示してSTOPするようにしています（リストA-1-1, A-1-2)．

パラメータ名	処　　　理
DISK	ドライブの種別を表す 8インチ　……………90H 5インチ2D　…………50H 5インチ2DD　………70H
SECTOR.SIZE	1セクタ当たりのバイト数(MS-DOSの場合) 8インチ　………1024バイト 5インチ　…………512バイト (5インチ2Dの場合は使用しない)
DRIVE.NUMBER	ドライブ種別ごとのドライブ番号 BASICでのドライブ番号ではなく,装置ごとに0,1,2,3の値
TRACK	トラック番号 8インチ　………………0～76 5インチ2D　…………0～39 5インチ2DD　………0～79
SURFACE	ヘッド番号 サイド0…………0 サイド1…………1
SECTOR	セクタ番号 8インチ　………1～8 5インチ　………1～9(8)
ADDRESS 1%	リードしたデータを転送するアドレス
ERROR 1	エラーステータスが格納される リードエラーが起こった場合,第8ビットがセットされ, 0～7ビットにBIOSのエラーステータスがはいる

図A-1-1　マシン語ルーチンへ渡されるパラメータ

ラベル名	処　　　理
*EXAMINE.DISK.FORMAT	・MS-DOSディスクのフォーマットを調べる ・ドライブ番号をもとにFATを読み出してFATIDを調べ,そのディスクのパラメータを得る
*READ.FAT	・FATを読み出す ・B0000H(V-RAM)からFATデータを格納する
*READ.DIRECTORY	・DIRECTORYを読み出す ・FATデータに続けてDIRECTORYデータを格納する
*SEARCH.DIRECTORY	・目的のファイルが存在するか調べる ・DIRECTORYを検索して与えられたファイル・ネームと一致するエントリがあれば,ファイルサイズと最初のクラスタ番号も調べる
*MAKE.LINK	・FATを検索して,与えられたファイルのあるクラスタを得る ・最初のクラスタ番号からそのファイルが記録されている全クラスタデータをCRASTER.LINK(I)に得る
*READ.DATA	・ファイルを読み込む ・CRASTER.LINK(I)に示されるクラスタを順次リードしてB0000Hから格納する

図A-1-2　主なサブルーチン

リスト A-1-1　トランスファー・プログラム

```
1000  'ms-dos to disk-basic file transfer (binary)
1010  CLEAR,&H1B00 : DEF SEG=&H1B00 : DEFINT B-Z : CLS 3
1020  CONSOLE 0,25,0,1: SCREEN 0,1,0,0: ADDRESS=0: SOFFSET=0
1030  FOR I=0 TO &H4E
1040  READ A$: A=VAL("&H"+A$):POKE ADDRESS+I,A
1050  NEXT
1060  DIM CRASTER.LINK(63): CRASTER.LINK(63)=0
1070  READ.SECTOR=&H0: BUF.SEGMENT=&HB000
1080  PRINT "   Floppy disk transfer 2/1 "
1090  PRINT
1100  PRINT " Set   dos-disk "
1110  INPUT " drive number";DRIVE.NUMBER
1120  GOSUB *EXAMINE.DISK.FORMAT
1130  GOSUB *READ.FAT
1140  GOSUB *READ.DIRECTORY
1150  GOSUB *DISPLAY.DIRECTORY
1160 *LOOP1
1170  INPUT " dos-filename";DNAME$
1180  N$=SPACE$(11): L=INSTR(DNAME$,"."): L1=LEN(DNAME$)-L
1190  IF L>9 OR L=0 OR L1>3 THEN *LOOP1
1200  MID$(N$,1,L-1)=DNAME$: MID$(N$,9,L1)=RIGHT$(DNAME$,L1)
1210  FOR I=1 TO LEN(N$): N2=ASC(MID$(N$,I,1))
1220    IF N2>=ASC("a") AND N2<=ASC("z")
        THEN MID$(N$,I,1)=CHR$(N2-&H20)
1230  NEXT
1240  DNAME$=N$
1250  INPUT " bas-filename";BNAME$
1260  PRINT " Ready(y/n)";: A$=INPUT$(1): PRINT A$
1270  IF A$="y" OR A$="Y" THEN ELSE *LOOP1
1280  GOSUB *SEARCH.DIRECTORY
1290  GOSUB *MAKE.LINK
1300  GOSUB *READ.DATA
1310  PRINT " Set basic-disk"
1320  INPUT " drive number";D.NUM
1330  FILE$=RIGHT$(STR$(D.NUM),1)+":"+BNAME$
1335  INPUT " offset";SOFFSET
1340  DEF SEG=BUF.SEGMENT
1350  IF F.SIZE!>&H7FFF THEN F.SIZE%=F.SIZE!-65536!
                          ELSE F.SIZE%=F.SIZE!
1360  BSAVE FILE$,SOFFSET,F.SIZE%-SOFFSET
1370  PRINT " Function complete!!"
1380 END
1390 *READ.FAT
1400  LSECTOR=1: ADDRESS=&H0
1410  FOR I=1 TO SECTORS.PER.FAT
1420  GOSUB *READ1: LSECTOR=LSECTOR+1
1430  ADDRESS=ADDRESS+SECTOR.SIZE
1440  NEXT : RETURN
1450 *READ1
1460  IF ADDRESS>&H7FFF THEN ADDRESS1%=ADDRESS-65536!
                         ELSE ADDRESS1%=ADDRESS
1470  SECTOR=LSECTOR MOD (SECTORS.PER.TRACK*2)+1
1480  IF SECTOR>SECTORS.PER.TRACK THEN SURFACE=1
      : SECTOR=SECTOR-SECTORS.PER.TRACK ELSE SURFACE=0
1490  TRACK=INT(LSECTOR¥SECTORS.PER.TRACK¥2)
1500 *READ2
1510  DEF SEG=&H1B00
```

N88-BASIC(86)ver3.0
を使用する場合は,
CLEAR,&h1E00:DEF SEG=&h1E00……
に変更する.

```
1520  IF D.MAX.TRACK=39 THEN SECTOR=2*SECTOR-1
1530  CALL READ.SECTOR(DISK,SECTOR.SIZE,DRIVE.NUMBER,TRACK,
SURFACE,SECTOR,ADDRESS1%,ERROR1)
1540  IF (ERROR1 AND &H100)=0 THEN RETURN
1550  PRINT "read error ";HEX$(ERROR1 AND &HFF): STOP
1560 *READ.CRASTER
1570  LSECTOR=CRASTER*SECTORS.PER.CRASTER+PRE.SECTORS
1580  FOR LOOP4=1 TO SECTORS.PER.CRASTER: GOSUB *READ1
1590  LSECTOR=LSECTOR+1
1600  ADDRESS=ADDRESS+SECTOR.SIZE
1610  NEXT: RETURN
1620 *READ.DIRECTORY
1630  LSECTOR=1+SECTORS.PER.FAT*2
1640  FOR I=1 TO DIRECTORY.ENTRY*32¥SECTOR.SIZE
1650    GOSUB *READ1: LSECTOR=LSECTOR+1
1660    ADDRESS=ADDRESS+SECTOR.SIZE
1670  NEXT: RETURN
1680 *DISPLAY.DIRECTORY
1690  DEF SEG=BUF.SEGMENT: D.BASE=SECTORS.PER.FAT*SECTOR.SIZE
1700  I=0: K=0
1710  WHILE I<DIRECTORY.ENTRY
1720    ADDRESS=D.BASE+I*32: D=PEEK(ADDRESS)
1730    IF D=0 THEN PRINT: RETURN
1740    IF D=&HE5 THEN 1850: ELSE K=K+1
1750    QNAME$=""
1760    FOR J=0 TO 7
1770      QNAME$=QNAME$+CHR$(PEEK(ADDRESS+J))
1780    NEXT
1790    QNAME$=QNAME$+" "
1800    FOR J=8 TO 10
1810      QNAME$=QNAME$+CHR$(PEEK(ADDRESS+J))
1820    NEXT
1830    PRINT USING "&            &   ";QNAME$;
1840    IF K MOD 5=0 THEN PRINT;
1850    I=I+1
1860  WEND: RETURN
1870 *EXAMINE.DISK.FORMAT
1880  D.MAX.TRACK=DSKF(DRIVE.NUMBER,0)
1890  IF D.MAX.TRACK=76 THEN SECTOR.SIZE=1024: DISK=&H90
                             : GOTO *DISK51
1900  SECTOR.SIZE=512
1910  IF D.MAX.TRACK=79 THEN DISK=&H70 ELSE DISK=&H50
1920 *DISK51
1930  DEF SEG=&H60: UC.DSK=PEEK(&H503)
1940  UC.DDCB=PEEK(&H508)+PEEK(&H509)*256
1950  FOR I=0 TO UC.DSK-1
1960  IF DRIVE.NUMBER=PEEK(UC.DDCB+I*20)
      THEN DRIVE.NUMBER=(PEEK(UC.DDCB+I*20+1) AND &HF)
      ELSE NEXT: PRINT "DRIVE NOT FOUND": STOP
1970  TRACK=0: SURFACE=0: SECTOR=2: ADDRESS1%=0
1980  GOSUB *READ2
1990  DEF SEG=BUF.SEGMENT: ID=PEEK(&H0)
2000  IF D.MAX.TRACK=76 AND ID=&HFE THEN SECTORS.PER.FAT=2
      : DIRECTORY.ENTRY=192:SECTORS.PER.TRACK=8:PRE.SECTORS=9
      : SECTORS.PER.CRASTER=1: RETURN
2010  IF D.MAX.TRACK=39 AND (ID=&HF9 OR ID=&HFB)
      THEN PRINT " Can not read!": STOP
2020  DIRECTORY.ENTRY=112: SECTORS.PER.CRASTER=2
2030  IF ID=&HF9 THEN PRE.SECTORS=10:
      SECTORS.PER.FAT=3: SECTORS.PER.TRACK=9: RETURN
```

```
2040  IF ID=&HFB THEN PRE.SECTORS=8:
        SECTORS.PER.FAT=2: SECTORS.PER.TRACK=8: RETURN
2050  D.MAX.TRACK=39
2060  IF ID=&HFD THEN PRE.SECTORS=8:
        SECTORS.PER.FAT=2: SECTORS.PER.TRACK=9: RETURN
2070  IF ID=&HFF THEN PRE.SECTORS=6:
        SECTORS.PER.FAT=1: SECTORS.PER.TRACK=8: RETURN
2080  PRINT 'fat id error': STOP
2090 *SEARCH.DIRECTORY
2100  DEF SEG=BUF.SEGMENT: D1=ASC(DNAME$)
2110  D.BASE=SECTORS.PER.FAT*SECTOR.SIZE:I=0
2120  WHILE I<DIRECTORY.ENTRY
2130    ADDRESS=D.BASE+I*32: D=PEEK(ADDRESS)
2140    IF D=&HE5 OR D<>D1 THEN *N1
2150    IF D=0 THEN *NOT.FOUND
2160    T=1
2170    WHILE T<11
2180     IF PEEK(ADDRESS+T)=ASC(MID$(DNAME$,T+1,1))
         THEN T=T+1 ELSE *N1
2190    WEND
2200    PRINT ' find !!' : F.ATTR=PEEK(ADDRESS+&HB)
2210    F.FIRST.CRASTER=PEEK(ADDRESS+&H1A)+PEEK(ADDRESS+&H1B)*256
2220    F.SIZE!=PEEK(ADDRESS+&H1C)+PEEK(ADDRESS+&H1D)*256
2230    IF PEEK(ADDRESS+&H1E) OR PEEK(ADDRESS+&H1F)
        THEN PRINT 'file too long': STOP
2240    RETURN
2250   *N1
2260    I=I+1
2270  WEND
2280 *NOT.FOUND
2290  PRINT 'file not found': STOP
2300 *MAKE.LINK
2310  I=0: CRASTER.LINK(I)=F.FIRST.CRASTER: DEF SEG=BUF.SEGMENT
2320  WHILE NOT (CRASTER.LINK(I)>=&HFF8 AND CRASTER.LINK(I)<=&HFFF)
2330    IF CRASTER.LINK(I)=&HFF7 OR CRASTER.LINK(I)=0 THEN
           PRINT 'craster abnormal': STOP
2340    ADDRESS=INT(CRASTER.LINK(I)*1.5): I=I+1
2350    F.LOW=PEEK(ADDRESS):F.HIGH=PEEK(ADDRESS+1)
2360    IF (CRASTER.LINK(I-1) MOD 2)=0
           THEN CRASTER.LINK(I)=(F.LOW+(F.HIGH AND &HF)*256)
           ELSE CRASTER.LINK(I)=INT((F.LOW+F.HIGH*256)/16)
2370  WEND
2380  MAX.LINK=I-1: RETURN
2390 *READ.DATA
2400  DEF SEG=BUF.SEGMENT: ADDRESS=&H0
2410  FOR I=0 TO MAX.LINK
2420    CRASTER=CRASTER.LINK(I):GOSUB *READ.CRASTER
2430  NEXT
2440  RETURN
2450 DATA 1E,53,B4,56,B5,03,C4,77,1C,26,8A,04,3C,90,74,08
2460 DATA B5,02,3C,70,74,02,B4,06,C4,77,10,26,8A,0C,C4,77
2470 DATA 14,26,0A,04,C4,77,0C,26,8A,34,C4,77,08,26,8A,14
2480 DATA C4,77,04,26,8B,2C,C4,77,18,26,FF,34,BB,00,B0,8E
2490 DATA C3,5B,CD,1B,8A,C4,9F,5B,1F,C4,37,26,89,04,CF
```

リスト A-1-2 マシン語部ソースリスト

```
;        read 1 sector subroutine 1/2
disk                equ     4*7
sector_size         equ     4*6
drive_number        equ     4*5
track               equ     4*4
surface             equ     4*3
sector              equ     4*2
address             equ     4*1
error1              equ     0
;
main      segment
          assume cs:main
          push    ds
          push    bx
          mov     ah,56h             ;read command
          mov     ch,3               ;1024byte/sector
          les     si,[bx+disk]
          mov     al,es:[si]
          cmp     al,90h             ;8inch?
          jz      std1
          mov     ch,2               ;512byte/sector
          cmp     al,70h             ;5inch2dd?
          jz      std1
          mov     ah,06h             ;5inch2d's read command
std1:     les     si,[bx+track]
          mov     cl,es:[si]
          les     si,[bx+drive_number]
          or      al,es:[si]
          les     si,[bx+surface]
          mov     dh,es:[si]
          les     si,[bx+sector]
          mov     dl,es:[si]
          les     si,[bx+address]
          mov     bp,es:[si]
          les     si,[bx+sector_size]
          push    es:[si]
          mov     bx,0B000H
          mov     es,bx
          pop     bx
          int     1bh
;
          mov     al,ah    ;error code to al
          lahf
          pop     bx
          pop     ds
          les     si,[bx+error1]
          mov     es:[si],ax
          iret
;
main      ends
;
          end
```

APPENDIX 2

MASMの高度な利用法

MASMは高い機能をもつアセンブラですが，本書では入門ということもあり，その機能をすべて使っているわけではありません．そこで本文中で説明しなかったいくつかの機能をここで紹介したいと思います．

○マクロ定義機能

何回か同じ処理を行うけれども，サブルーチンにするほどではないというような場合があります．その時その処理をマクロ定義して気のきいた名前をつけておけば，わかりやすく，すっきりしたソース・プログラムにすることができます．リストA-2-1はROM内ルーチンのコールをマクロ定義した例です．

リスト A-2-1 マクロ定義例(ソース)

```
;
;         masm-1
;
main      segment
          assume  cs:main
line_in   macro
          push    ax
          push    ds
          mov     ax,60h
          mov     ds,ax
          mov     di,82
          int     0c4h
          pop     ds
          pop     ax
          endm
;
;             :
;             :
;             :
          line_in
          mov     bx,202h
;             :
          line_in
;             :
main      ends
          end
              :
```

line＿inの処理定義

line＿inマクロの参照

このマクロ定義はいくつかの命令の集まりをひとつの名前で代表させるもので，以後，名前がでてくる度に定義された命令がそこに展開されて出力されます．サブルーチンのように，処理を行う部分が1つあってそれをCALLするのではなく，その処理部分が参照されるごとに，オブジェクトになるわけです．したがって何度もその処理を行うと，オブジェクトが長大なものになるので注意が必要です（リスト A-2-2）．

リスト A-2-2 マクロ定義例(リスト)

```
                         ;
                         ;       masm-1
                         ;
0000                     main    segment
                                 assume   cs:main
                         line_in macro
                                 push     ax
                                 push     ds
                                 mov      ax,60h
                                 mov      ds,ax
                                 mov      di,82
                                 int      0c4h
                                 pop      ds
                                 pop      ax
                                 endm
                         ;
                         ;
                         ;
                         ;            :
                                      :
                                 line_in
0000   50                (+)     push     ax
0001   1E                 +      push     ds
0002   B8 0060            +      mov      ax,60h
0005   8E D8              +      mov      ds,ax
0007   BF 0052            +      mov      di,82
000A   CD C4              +      int      0c4h
000C   1F                 +      pop      ds
000D   58                 +      pop      ax
000E   BB 0202                   mov      bx,202h
                         ;            :
                                 line_in
0011   50                 +      push     ax
0012   1E                 +      push     ds
0013   B8 0060            +      mov      ax,60h
0016   8E D8              +      mov      ds,ax
0018   BF 0052            +      mov      di,82
001B   CD C4              +      int      0c4h
001D   1F                 +      pop      ds
001E   58                 +      pop      ax
                         ;            :
001F                     main    ends
                                 end
```

MACROの展開であることを示す

line _ inの処理が展開される

上と同様再び展開される

マクロ定義の中にラベルを宣言して使う場合，LOCAL擬似命令により，そのラベルを局所的なシンボルとして扱う必要があります．なぜなら，このようにしないと2回以上展開した時，重複

定義になるからです．

マクロ定義には定義時の指定により，パラメータを渡すことができます．リストA-2-3ではBASICのPRINT文のように扱えるマクロ定義の例です．定義でのstringsはダミー・パラメータで，展開される時には実際の値に置き換わります．

リスト A-2-3 パラメータを渡す例

```
;
;       masm-3
;
main    segment
        assume  cs:main                       print MACRO命令の定義
print   macro   strings   ―――――――――――― stringsというダミーパラメータを設定する
        local   s1,s2,s3  ―――――――――――― s1, s2, s3 は展開中にローカルなラベルとして扱われる
        push    si
        push    cx
        push    ax
        push    di
        push    ds
        mov     ax,60h
        mov     ds,ax
        mov     si,offset s1
        mov     cx,offset s2- offset s1― 文字列の長さを得る
        jcxz    s2
s3:     mov     di,61         ┐
        mov     al,cs:[si]    │ 1文字出力のシステムコール
        inc     si            │
        int     0c4h          ┘
        loop    s3
        jmp     short s2
s1:     db      strings   ―――――――――――― 実際のパラメータがstringsと置き換わる
s2:     pop     ds
        pop     di
        pop     ax
        pop     cx
        pop     si
        endm
;
;
;
        print 'Machine Friend'――――――――― printに続けてパラメータを書く
+       push    si
+       push    cx
+       push    ax
+       push    di
+       push    ds
+       mov     ax,60h
+       mov     ds,ax
+       mov     si,offset ??0000                       ┐ ??で始まるラベルは
+       mov     cx,offset ??0001- offset ??0000        ┘ MASM内で順次生成されるため複数の展開で重複定義にならない
+       jcxz    ??0001
+ ??0002: mov   di,61
+       mov     al,cs:[si]
+       inc     si
+       int     0c4h
+       loop    ??0002
+       jmp     short ??0001
```

```
+ ??0000: db      'Machine Friend'
+ ??0001: pop     ds
+         pop     di
+         pop     ax
+         pop     cx
+         pop     si
  ;
          print<0dh,0ah,'moon branch'>
+         push    si
+         push    cx
+         push    ax
+         push    di
+         push    ds
+         mov     ax,60h
+         mov     ds,ax
+         mov     si,offset ??0003
+         mov     cx,offset ??0004- offset ??0003
+         jcxz    ??0004
+ ??0005: mov     di,61
+         mov     al,cs:[si]
+         inc     si
+         int     0c4h
+         loop    ??0005
+         jmp     short ??0004
+ ??0003: db      0dh,0ah,'moon branch'
+ ??0004: pop     ds
+         pop     di
+         pop     ax
+         pop     cx
+         pop     si
  ;
  ;                 ;
  main    ends
          end
```

パラメータが置き換わっている

パラメータ内に“，”カンマを含む時には不等号で1つのパラメータの区切りを示す

print MACROの2度目の展開．ラベルに新たな名前がついて展開されていることに注意

マクロ機能にはそのほかに，リピート擬似命令を用いるものがあります．これは上のマクロ定義と異なって定義部分がなく，そのまま他の命令とともに使われます．

○条件アセンブル機能

条件アセンブルとは，特定の条件を調べてそれにしたがった部分のアセンブルを行うものです．この機能はIF~ELSE~ENDIFの擬似命令によって実現されます．この機能はたとえばリストA-2-4のように，MS-DOS上とBASIC上の両方で動かしたいプログラムを作成する時に便利です．systemのEQUによって設定する値を変えてアセンブルすれば，システムに応じたルーチンを得ることができます．

リスト A-2-4 条件アセンブルの例 1

```
;
;         masm-4
;
basic     equ     0ffffh
msdos     equ     not basic
```

IF文で使われる真偽の設定
basicは真，msdosは偽

```
;
system  equ     msdos ———— systemは偽になる
;
main    segment
        assume  cs:main
outchr:
        ;al : chr data
;
    if  system
        push    ds
        push    di
        mov     di,61
        mov     ds,di              systemは偽のためこの部分はアセンブルされない
        int     0c4h
        pop     di
        pop     ds
        ret
;
    else
        ;ms-dos
        push    ax
        push    dx
        mov     ah,2
        mov     dl,al              この部分がアセンブルされコードが生成される
        int     21h
        pop     dx
        pop     ax
        ret
;
    endif
main    ends
        end
```

IF 命令は，マクロ定義中でも使用することができます．リスト A-2-5 は 1 命令で複数のレジスタの値を PUSH，POP するような命令の定義例です．

リスト A-2-5 条件アセンブルの例 2

```
;
;       masm-5
;
main    segment
        assume cs:main
apush   macro   a,b,c,d,e,f,g,h,i,j,k,l ———— 12個までパラメータを与えることができる
        irp     n,<a,b,c,d,e,f,g,h,i,j,k,l>
        ifnb    <n>                パラメータがブランクでない時には
        push    n                  PUSH命令を生成する
        endif
        endm
        endm
;
apop    macro   a,b,c,d,e,f,g,h,i,j,k,l
        irp     n,<a,b,c,d,e,f,g,h,i,j,k,l>
        ifnb    <n>
        pop     n                  irpは，繰り返しマクロでパラメータが順
        endif                      次nに与えられていく
        endm
```

```
          endm                                   ┐
 ;                                               ┐
          apush     ax,di,cx
+         push      ax
+         push      di
+         push      cx
 ;
          apop      ds,es
+         pop       ds
+         pop       es
 ;
main      ends
 ;
          end
```

ifnbがあるためパラメータの数が12個より，少なくても正常に動く

○分割アセンブル機能

MASMはオブジェクト・コードとしてリロケータブルな形式のコードを出力します．このコードにはマシン語本体とリロケートするための情報が含まれておりMS-DOSのLINK（リンカ）によってオブジェクトを結合することができます．つまり，あるプログラムを作る時，それをいくつかのモジュールに分割して作成，アセンブルしておいて，実行時に結合（リンク）すればよいのです．この機能を用いることによってプログラムのパーツ化（ライブラリ）が可能になり，保守の容易さや多人数での開発の点で有利になります．

リストA-2-6,7,8にその一例を示します．これは，入力と出力ルーチンを別に作っておき，最後にメイン・ルーチンと結合して使うという例です．リスト中，EXTRNはその名前が他のモジュールで定義されていることを示す擬似命令で，PUBLICはその名前が他のモジュールで利用できるようにする擬似命令です．

リスト A-2-6 分割アセンブルの例 1

```
 ;
 ;        masm-6
 ;
key       segment
          assume    cs:key
          public    line_in ──── line_inというラベルが外部のモジュールで参照できるようになる
 line_in:
          push      di
          mov       di,60h
          mov       ds,di
          mov       di,82
          int       0c4h
          mov       si,202h
          pop       di
          ret
 ;
key       ends
 ;
          end
```

ライン入力ルーチン
入力データは60H：202Hからのバッファにはいる

リスト A-2-7 分割アセンブルの例 2

```
;
;       masm-7
;
crt     segment
        assume  cs:crt
        public  line_out   ; line_out というラベルが外部モジュールで参照できるようにする
line_out        proc near
        ;cx: data length
        ;bx: buffer offset
        ;es: buffer seg
        push    ax         ; ┐ ES:BXで指定するバッファの内容をCXバイト出力する
        push    di
        push    ds
        mov     ax,60h
        mov     ds,ax
        jcxz    a2
a1:     mov     di,61
        mov     al,es:[bx]
        int     0c4h
        inc     bx
        loop    a1
a2:     pop     ds
        pop     di
        pop     ax
        ret                ; ┘
line_out        endp
crt     ends
;
        end
```

リスト A-2-8 分割アセンブルの例 3

```
;
;       masm-8
;
main    segment
        assume  cs:main
        extrn   line_in:near,line_out:near   ; これらのラベルが外部のモジュールにあることを示す
toupper:call    line_in    ; 外部モジュールへのコール
        push    ds         ; ┐ 入力したデータを大文字に変換して buff1 に転送する
        push    bx
        mov     ax,cs
        mov     es,ax
        mov     di,offset buff1
        push    cx
        cld
        jcxz    a1
a2:     dec     cx
        lodsb
        cmp     al,'a'
        jc      a3
        cmp     al,'z'+1
        jnc     a3
        and     al,11011111b
a3:     stosb
        jmp     short a2
;
```

```
a1:     pop     cx
        mov     bx,offset buff1
        call    line_out            buff1の内容を出力する
        pop     bx                  外部モジュールへコール
        pop     ds
        iret
;
buff1   db      256 dup(?)
;
main    ends
;
        end
```

○擬似命令一覧

図 A-2-1 に MASM で使用される擬似命令の一覧を示します．

メモリ構成擬似命令	
ASSUME	セグメント・レジスタがどのセグメントを指すかをアセンブラに知らせる
ASSUME NOTHING	暗黙指定のセグメント・レジスタをなくし，すべて明示により動作する
COMMENT	区切り記号ではさまれた文字列をコメントと見なすようにアセンブラに知らせる
DB, DW, DD, DQ, DT	それぞれの型の変数を定義する
END	ソース・ステートメントの終わり
EQU	名前に数値を割り当てる
=	再定義できる EQU 擬似命令
EXTRN	シンボルが外部定義であることを示す
PUBLIC	シンボルが外部のモジュールで使用されてもよいことを示す
LABEL	コード・セグメントに仮定されているアドレスに名前を付ける
NAME	モジュールの名前を定義する
PROC ～ ENDP	モジュール本体を PROC ～ ENDP でかこむ
.RADIX	入力する数値の基数を定義する
RECORD	ビット単位でフィールドを定義する
GROUP	セグメントの集合に名前を付ける
SEGMENT ～ ENDS	この擬似命令にはさまれたソースを，このセグメント内のコードとする
EVEN	ロケーションカウンタの値をワード - バウンダリに合せる
ORG	ロケーションカウンタの値をセットする
STRUC ～ ENDS	バイト単位でストラクチャを定義する
INCLUDE	別のソースファイルをこの位置に挿入する

マクロ擬似命令	
EXITM	残りのマクロを展開せず，マクロを終了する
IRP	不定回数の繰返し
IRPC	不定回数の文字繰返し
LOCAL	マクロ内でローカルなシンボルを作成する
MACRO－ENDM	MACRO から ENDM 内がマクロ定義である
PURGE	それ以降のマクロ定義を削除する
REPT	REPT と ENDM 間のマクロを繰返す

条件アセンブル擬似命令	
ELSE	反対条件のときアセンブルされるソースを定義
ENDIF	条件アセンブルブロックを終了させる
IF	式がゼロ以外ならアセンブルを行う
IFB	引数がブランクまたは空白ならアセンブルを行う
IFDEF	シンボルが定義されているとアセンブルを行う
IFDIF	第１引数と第２引数が違えばアセンブルを行う
IFE	式がゼロならアセンブルを行う
IFIDN	第１引数と第２引数が同じならアセンブルを行う
IFNB	引数がブランクまたは空白でなければアセンブルを行う
IFNDEF	シンボルが定義されていないとアセンブルを行う
IF1	パス１の最中，アセンブルを行う
IF2	パス２の最中，アセンブルを行う

リスト制御擬似命令	
.CREF	クロスリファレンスを作成する
.LALL	すべてのマクロを展開し表示する
.LFCOND	条件リスティング（偽なら表示）
.LIST	すべてのリストを作成する
%OUT	ソースリストをアセンブル中に画面に出力する
PAGE	リストを改ページする
.SALL	すべてのマクロを展開しない
.SFCOND	条件リスティング（真なら表示）
SUBTTL	サブタイトルを定義する
.TFCOND	混在の条件を反転する
TITLE	各ページの最初のタイトルを定義する
.XALL	マクロで生成されたコードだけを表示する
.XCREF	クロスリファレンスを抑止する
.XLIST	すべてのリストの作成を中止する

（"16bit Personal Computer" より引用）

図 A-2-1　擬似命令一覧

索引

参考文献

〔1〕 "iAPX 86 マクロアセンブリ言語 プログラミングマニュアル"
発行 インテルジャパン（発売CQ出版）

〔2〕 "PC-9801 システム解析(上)(下)"
アスキー

〔3〕 "PC-8801 マシン語入門"
アスキー

〔4〕 "Threaded Interpretive Languages"〈BYTE Books〉(Loeliger)
マグロウヒル

〔5〕 "An Architectural Trail to Threaded-Code Systems" (Peter M. Kogge)

協力 羽根潤・加賀美雄二

PC-9801 マシン語入門
アスキー・システム・バンク(PC98#2)

1985年4月15日　初版発行
1990年4月11日　第1版第15刷発行
定価2,580円(本体2,505円)

著　者　岩瀬正幸・藤井敬雄

発行者　塚本慶一郎
発行所　株式会社アスキー
　〒107 東京都港区南青山6-11-1スリーエフ南青山ビル
振　替　東京4-161144
TEL (03)486-7111(大代表)
情報TEL (03)498-0299(ダイヤルイン)
出版営業部TEL (03)486-1977(ダイヤルイン)

編集担当　高橋　匠
表紙担当　郷　啓子
印刷　壮光舎印刷株式会社

ISBN4-87148-775-X

●11091／11435

折込み付録
8086 インストラクション表

アスキー・システム・バンク(PC98#2)
PC-9801 マシン語入門
岩瀬正幸・藤井敬雄共著

アスキー出版局

機能別 8086 命令一覧表

オペランド・タイプ

名　前	説　　明
reg	8/16ビット汎用レジスタ
reg 8	8ビット汎用レジスタ
reg 16	16ビット汎用レジスタ
mem	8/16ビット・メモリロケーション
mem 8	8ビット・メモリロケーション
mem 16	16ビット・メモリロケーション
mem 32	32ビット・メモリロケーション
imm 8	0～FFHの範囲の定数
imm 16	0～FFFFHの範囲の定数
acc	レジスタ AX または AL
sreg	セグメント・レジスタ
source-table	256バイト翻訳テーブルの名称
dst-string	レジスタ DI でアドレスされるストリングの名称
src-string	レジスタ SI でアドレスされるストリングの名称
near-proc	現在のコード・セグメント内のプロシージャ
far-proc	別のコード・セグメント内のプロシージャ
near-label	現在のコード・セグメント内のラベル
short-label	命令の終りから－128～＋127バイトの範囲のラベル
far-label	別のコード・セグメント内のラベル
memptr 16	制御が移されようとしている現在のコード・セグメント内のロケーションのオフセットを含むワード
memptr 32	制御が移されようとしている別のコード・セグメント内のロケーションのオフセットとセグメントベース・アドレスを含むダブルワード
regptr 16	制御が移されようとしている現在のコード・セグメント内のロケーションのオフセットを含む16ビット汎用レジスタ
repeat	ストリング命令のリピート・プレフィクス
pop-value	スタックから捨てるバイト数（0～64K，通常は偶数）
ext-op	外部プロセッサによって使用されるために命令中にエンコードされるイミディエイト値（0～63）

オペコード・フィールド

名前	説明
W	ワード/バイト・フィールド（0～1）
reg	レジスタ・フィールド（000～111）
r/m	レジスタ/メモリ・フィールド（000～111）
mod	モード・フィールド（00～10）
S：W	S：W＝01のとき data＝16ビット，それ以外は data＝8ビット S：W＝11のときバイトデータのサインが拡張されて16ビット・オペランドを作る
XXX	ESCオペコードの初めの3ビット
YYY	ESCオペコードの2番目の3ビット

名前	説明
AX	アキュムレータ（16ビット）
AH	アキュムレータ（上位バイト）
AL	アキュムレータ（下位バイト）
BX	レジスタB（16ビット）
CX	レジスタC（16ビット）
CL	レジスタC（下位バイト）
DX	レジスタD（16ビット）
SP	スタック・ポインタ（16ビット）
IP	インストラクション・ポインタ（16ビット）
FLAGS	フラグ用レジスタ空間（16ビット）
DI	ディスティネーション・インデクス・レジスタ
SI	ソース・インデクス・レジスタ
CS	コード・セグメント・レジスタ
ES	エクストラ・セグメント・レジスタ
AF	補助キャリー・フラグ
CF	キャリー・フラグ
PF	パリティ・フラグ
SF	サイン・フラグ
ZF	ゼロ・フラグ
DF	ディレクション・フラグ
IF	割込み許可フラグ
OF	オーバーフロー・フラグ
TF	トラップ・フラグ
(…)	(　　) 内で示されるレジスタまたはメモリ（ダイレクト）の内容
(…)	(　　) 内で示されるメモリ（レジスタ・インダイレクト）の内容
：(コロン)	2個の8ビット・バイトの連続アドレス
disp	ディスプレースメント（8/16ビット）
temp	テンポラリ・レジスタ（8ビット）
tmpcf	テンポラリ・キャリー・フラグ（1ビット）
seg	イミディエイト・セグメント・データ（16ビット）
offset	イミディエイト・オフセット・データ（16ビット）
EXT	拡張バイトまたはワード
LOW	下位バイトまたはワード
←	転送方向
＋	加算
－	減算
＊	乗算
/	除算
%	モデュロ
&	論理積
∨	論理和
∀	排他的論理和
××H	16進2桁の数値
××××H	16進4桁の数値

フラグの動作

名　前	説　　明
(ブランク)	変化なし
0	0にクリアされる
1	1にセットされる
×	結果に従ってセットまたはクリアされる
U	不　定
R	以前に退避した値がリストアされる

メモリ・アドレッシング

r/m ＼ mod	0 0	0 1	1 0
0 0 0	(BX)＋(SI)	(BX)＋(SI)＋disp 8	(BX)＋(SI)＋disp 16
0 0 1	(BX)＋(DI)	(BX)＋(DI)＋disp 8	(BX)＋(DI)＋disp 16
0 1 0	(BP)＋(SI)	(BP)＋(SI)＋disp 8	(BP)＋(SI)＋disp 16
0 1 1	(BP)＋(DI)	(BP)＋(DI)＋disp 8	(BP)＋(DI)＋disp 16
1 0 0	(SI)	(SI)＋disp 8	(SI)＋disp 16
1 0 1	(DI)	(DI)＋disp 8	(DI)＋disp 16
1 1 0	DIRECT ADDRESS	(BP)＋disp 8	(BP)＋disp 16
1 1 1	(BX)	(BX)＋disp 8	(BX)＋disp 16

8/16ビット汎用レジスタの選択

reg or r/m*	W＝0	W＝1
0 0 0	A L	A X
0 0 1	C L	C X
0 1 0	D L	D X
0 1 1	B L	B X
1 0 0	A H	S P
1 0 1	C H	B P
1 1 0	D H	S I
1 1 1	B H	D I

＊ r/mはmodのない場合

セグメント・レジスタの選択

sreg	
0 0	E S
0 1	C S
1 0	S S
1 1	D S

クロック

メモリ・オペランドの場合，クロック数がアドレシング・モードにより異なるため，命令の説明のクロック数の項で＋EAと書かれている分については，次に示す数値を適用する

○ダイレクト16ビット・オフセット・アドレス　6
○ベースまたはインデクス・レジスタによるインダイレクト　5
○ディスプレースメントを伴ったベースまたはインデクス・レジスタによるインダイレクト　9
○インデクス・レジスタとベース・レジスタとの和によるインダイレクト　7 or 8
○ディスプレースメントを伴ったインデクス・レジスタとベース・レジスタとの和によるインダイレクト　11 or 12

これらの値は偶数アドレスに存在する8ビット・オペランドまたは16ビット・オペランドに対するもので，奇数アドレスに存在する16ビット・オペランドではさらに4クロック加える
またセグメント・オーバライドには2クロックを加える

分類	ニーモニック	オペランド	オペレーション・コード 76543210	オペレーション・コード 76543210	バイト数	クロック数	オペレーション	A	C	O	P	S	Z
[illegible]	MOV	reg, reg	1000100W	11 reg r/m	2	2	(reg)←(reg)						
		mem, reg	1000100W	mod reg r/m	2-4	9+EA	(mem)←(reg)						
		reg, mem	1000101W	mod reg r/m	2-4	8+EA	(reg)←(mem)						
		mem, imm	1100011W	mod000 r/m	3-6	10+EA	(mem)←(data)						
		reg, imm	1011W reg		2-3	4	(reg)←(data)						
		acc, mem	1010000W		3	10	if W=0 (AL)←(addr) if W=1 (AX)←(addr+1 : addr)						
		mem, acc	1010001W		3	10	if W=0 (addr)←(AL) if W=1 (addr+1 : addr)←(AX)						
		reg16, sreg	10001110	110sreg r/m	2	2	(sreg)←(reg16)						
		sreg, mem	10001110	mod0sreg r/m	2-4	8+EA	(sreg)←(mem)						
		reg16, sreg	10001100	110sreg r/m	2	2	(reg16)←(sreg)						
		mem, reg	10001100	mod0sreg r/m	2-4	9+EA	(mem)←(sreg)						
	PUSH	mem	11111111	mod110 r/m	2-4	16+EA	(SP)←(SP)−2 ((SP)+1 (SP))←(mem)						
		reg	01010 reg		1	10	(SP)←(SP)−2 ((SP)+1 (SP))←(reg)						
		sreg	000sreg110		1	10	(SP)←(SP)−2 ((SP)+1 (SP))←(sreg)						
	POP	mem	10001111	mod000 r/m	2-4	17+EA	(mem)←((SP)+1 (SP)) (SP)←(SP)+2						
		reg	01011 reg		1	8	(reg)←((SP)+1 (SP)) (SP)←(SP)+2						
		sreg	000sreg111		1	8	(sreg)←((SP)+1 (SP)) (SP)←(SP)+2						
	XCHG	reg, reg	1000011W	11 reg r/m	2	4	(reg)↔(reg)						
		mem, reg	1000011W	mod reg r/m	2-4	17+EA	(reg)↔(mem)						
		AX, reg16	10010 reg		1	3	(reg16)↔(AX)						
アキュムレータ転送命令	IN	acc, imm8	1110010W		2	10	ifW=0(AL)←(imm8) ifW=1(AX)←(imm8+1 imm8)						
		acc, DX	1110110W		1	8	ifW=0(AL)←((DX)) ifW=1(AX)←((DX)+1 (DX))						
	OUT	imm8, acc	1110011W		2	10	ifW=0(imm8)←(AL) ifW=1(imm8+1 imm8)←(AX)						
		DX, acc	1110111W		1	8	ifW=0((DX))←(AL) ifW=1((DX)+1 (DX))←(AX)						
	XLAT	source table	11010111		1	11	(AL)←((BX)+(AL))						
アドレス転送命令	LEA	reg16, mem16	10001101	mod reg r/m	2-4	2+EA	(reg16)←mem16						
	LDS	reg16, mem32	11000101	mod reg r/m	2-4	16+EA	(reg16)←(mem32) (DS)←(mem32+2)						
	LES	reg16, mem32	11000100	mod reg r/m	2-4	16+EA	(reg16)←(mem32) (ES)←(mem32+2)						
フラグ・レジスタ転送命令	LAHF		10011111		1	4	(AH)←(SF) (ZF) X (AF) X (PF) X (CF)						
	SAHF		10011110		1	4	(SF) (ZF) X (AF) X (PF) X (CF)←(AH)	×	×		×	×	×
	PUSHF		10011100		1	10	(SP)←(SP)−2 ((SP)+1 (SP))←FLAGS						
	POPF		10011101		1	8	FLAGS←((SP)+1 (SP)) (SP)←(SP)+2	R	R	R	R	R	R
加算命令	ADD	reg, reg	0000000W	11 reg r/m	2	3	(reg)←(reg)+(reg)	×	×	×	×	×	×
		mem, reg	0000000W	mod reg r/m	2-4	16+EA	(mem)←(mem)+(reg)	×	×	×	×	×	×
		reg, mem	0000001W	mod reg r/m	2-4	9+EA	(reg)←(reg)+(mem)	×	×	×	×	×	×
		reg, imm	100000SW	11000 r/m	3-4	4	(reg)←(reg)+data	×	×	×	×	×	×
		mem, imm	100000SW	mod000 r/m	3-6	17+EA	(mem)←(mem)+data	×	×	×	×	×	×
		acc, imm	0000010W		2-3	4	ifW=0 (AL)←(AL)+data ifW=1 (AX)←(AX)+data	×	×	×	×	×	×
	ADC	reg, reg	0001000W	11 reg r/m	2	3	(reg)←(reg)+(reg)+(CF)	×	×	×	×	×	×
		mem, reg	0001000W	mod reg r/m	2-4	16+EA	(mem)←(mem)+(reg)+(CF)	×	×	×	×	×	×
		reg, mem	0001001W	mod reg r/m	2-4	9+EA	(reg)←(reg)+(mem)+(CF)	×	×	×	×	×	×
		reg, imm	100000SW	11010 r/m	3-4	4	(reg)←(reg)+data+(CF)	×	×	×	×	×	×
		mem, imm	100000SW	mod010 r/m	3-6	17+EA	(mem)←(mem)+data+(CF)	×	×	×	×	×	×
		acc, imm	0001010W		2-3	4	ifW=0 AL←AL (AL)←(AL)+data+(CF) ifW=1 AX←AX (AX)←(AX)+data+(CF)	×	×	×	×	×	×
	INC	reg8	1111111W	11000 r/m	2	3	(reg8)←(reg8)+1	×		×	×	×	×
		mem	1111111W	mod000 r/m	2-4	15+EA	(mem)←(mem)+1	×		×	×	×	×
		reg16	01000 reg		1	2	(reg16)←(reg16)+1	×		×	×	×	×

命令群	ニーモニック	オペランド	オペレーション・コード 76543210	76543210	バイト数	クロック数	オペレーション	フラグ A	C	O	P	S	Z
算術演算命令 加算命令	AAA		00110111		1	4	if ((AL)&0FH)>9or(AF)=1then(AL)←(AL)+6. (AH)←(AH)+1,(AF)←1,(CF)←(AF),(AL)←(AL)&0FH	×	×	U	U	U	U
	DAA		00100111		1	4	if ((AL)&0FH)>9or(AF)=1 then (AL)←(AL)+6, (CF)←(AF)∨(CF), (AF)←1 if (AL)>9FHor(CF)=1 then (AL)←(AL)+60H, (CF)←1	×	×	U	×	×	×
減算命令	SUB	reg,reg	0010100W	11 reg r/m	2	3	(reg)←(reg)−(reg)	×	×	×	×	×	×
		mem,reg	0010100W	mod reg r/m	2-2	16+EA	(mem)←(mem)−(reg)	×	×	×	×	×	×
		reg,mem	0010101W	mod reg r/m	2-4	9+EA	(reg)←(reg)−(mem)	×	×	×	×	×	×
		reg,imm	100000SW	11101 r/m	3-4	4	(reg)←(reg)−data	×	×	×	×	×	×
		mem,imm	100000SW	mod101 r/m	3-6	17+EA	(mem)←(mem)−data	×	×	×	×	×	×
		acc,imm	0010110W		2-3	4	ifW=0 (AL)←(AL)−data ifW=1 (AX)←(AX)−data	×	×	×	×	×	×
	SBB	reg,reg	0001100W	11 reg r/m	2	3	(reg)←(reg)−(reg)−(CF)	×	×	×	×	×	×
		mem,reg	0001100W	mod reg r/m	2-4	16+EA	(mem)←(mem)−(reg)−(CF)	×	×	×	×	×	×
		reg,mem	0001101W	mod reg r/m	2-4	9+EA	(reg)←(reg)−(mem)−(CF)	×	×	×	×	×	×
		reg,imm	100000SW	11011 r/m	3-4	4	(reg)←(reg)−data−(CF)	×	×	×	×	×	×
		mem,imm	100000SW	mod011 r/m	3-6	17+EA	(mem)←(mem)−data−(CF)	×	×	×	×	×	×
		acc,imm	0001110W		2-3	4	ifW=0 (AL)←(AL)−data−(CF) ifW=1 (AX)←(AX)−data−(CF)	×	×	×	×	×	×
	DEC	reg8	11111110	11001 r/m	2	3	(reg8)←(reg8)−1	×		×	×	×	×
		mem	1111111W	mod001 r/m	2-4	15+EA	(mem)←(mem)−1	×		×	×	×	×
		reg16	01001 reg		1	2	(reg16)←(reg16)−1	×		×	×	×	×
	NEG	reg	1111011W	11011 r/m	2	3	(reg)←0−(reg)	×	×	×	×	×	×
		mem	1111011W	mod011 r/m	2-4	16+EA	(mem)←0−(mem)	×	×	×	×	×	×
	CMP	reg,reg	0011100W	11 reg r/m	2	3	(reg)−(reg)	×	×	×	×	×	×
		mem,reg	0011100W	mod reg r/m	2-4	9+EA	(mem)−(reg)	×	×	×	×	×	×
		reg,mem	0011101W	mod reg r/m	2-4	9+EA	(reg)−(mem)	×	×	×	×	×	×
		reg,imm	100000SW	11111 r/m	3-4	4	(reg)−data	×	×	×	×	×	×
		mem,imm	100000SW	mod111 r/m	3-6	10+EA	(mem)−data	×	×	×	×	×	×
		acc,imm	0011110W		2-3	4	if W=0 (AL)−data if W=1 (AX)−data	×	×	×	×	×	×
	AAS		00111111		1	4	if ((AL)&0AH)>9 or (AF)=1 then (AL)←(AL)−6, (AH)←(AH)−1, (AF)←1 (CF)←(AF), (AL)←(AL)&0FH	×	×	U	U	U	U
	DAS		00101111		1	4	if ((AL)&0FH)>9 or (AF)=1 then (AL)←(AL)−6, (CF)←(AF)∨(CF), (AF)←1 if (AL)>9FH or (CF)=1 then (AL)←(AL)−60H, (CF)←1	×	×	U	×	×	×
乗算命令	MUL	reg8	11110110	11100 r/m	2	70-77	(AX)←(AL)＊(reg8) EXT=AH if(EXT)=0then(CF)←0 else(CF)←1;(OF)←(CF)	U	×	×	U	U	U
		mem8	11110110	mod100 r/m	2-4	(76-83) +EA	(AX)←(AL)＊(mem8) EXT=AH if(EXT)=0then(CF)←0 else(CF)←1;(OF)←(CF)	U	×	×	U	U	U
		reg16	11110111	11100 r/m	2	118-133	(DX:AX)←(AX)＊(reg16) EXT=DX if(EXT)=0then(CF)←0 else(CF)←1,(OF)←(CF)	U	×	×	U	U	U
		mem16	11110111	mod100 r/m	2-4	(124-139) +EA	(DX:AX)←(AX)＊(mem16) EXT=DX if(EXT)=0then(CF)←0 else(CF)←1;(OF)←(CF)	U	×	×	U	U	U
	IMUL	reg8	11110110	11101 r/m	2	80-98	(AX)←(AL)＊(reg8) EXT=AH, LOW=AL if(EXT)=sign extension of(LOW)then(CF)←0 else (CF)←1, (OF)←(CF)	U	×	×	U	U	U
		mem8	11110110	mod101 r/m	2-4	(86-104) +EA	(AX)←(AL)＊(mem8) EXT=AH, LOW=AL if(EXT)=sign extension of(LOW)then(CF)←0 else(CF)←1; (OF)←(CF)	U	×	×	U	U	U
		reg16	11110111	11101 r/m	2	128-154	(DX:AX)←(AX)＊(reg16) EXT=DX, LOW=AX if(EXT)=sign extension of(LOW)then(CF)←0 else(CF)←1, (OF)←(CF)	U	×	×	U	U	U
		mem16	11110111	mod101 r/m	2-4	(134-160) +EA	(DX:AX)←(AX)＊(mem16) EXT=DX, LOW=AX if(EXT)=sign extension of(LOW)then(CF)←0 else(CF)←1; (OF)←(CF)	U	×	×	U	U	U
	AAM		11010100	00001010	2	83	(AH)←(AL)/0AH (AL)←(AL)%0AH	U	U	U	×	×	×
除算命令	DIV	reg8	11110110	11110 r/m	2	80-90	(temp)←(AX) if(temp)/(reg8)>FFH then (SP)←(SP)−2, ((SP)+1:(SP))←FLAGS. (IF)←0, (TF)←0, (SP)←(SP)−2, ((SP)+1:(SP))←(CS), (CS)←(2)	U	U	U	U	U	U

命令群	ニーモニック	オペランド	オペレーション・コード 7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	バイト数	クロック数	オペレーション	フラグ A	C	O	P	S	Z
算術演算命令 除算命令	DIV	reg8	11110110	11110 r/m	2	89-90	(SP)←(SP)−2, ((SP)+1:(SP))←(IP), (IP)←(0) else (AL)←(temp)/(reg8) (AH)←(temp)%(reg8)	U	U	U	U	U	U
		mem8	11110110	mod110 r/m	2-4	(86-96) +EA	(temp)←(AX) if(temp)/(mem8)>FFH then (SP)←(SP)−2,((SP)+1:(SP))←FLAGS,(IF)←0,(TF)←0 (SP)←(SP)−2,((SP)+1:(SP))←(CS), (CS)←(2) (SP)←(SP)−2,((SP)+1:(SP))←(IP), (IP)←(0) else (AL)←(temp)/(mem8), (AH)←(temp)%(mem8)	U	U	U	U	U	U
		reg16	11110111	11110 r/m	2	144-162	(temp)←(DX:AX) if(temp)/(reg16)>FFFFH then (SP)←(SP)−2,((SP)+1:(SP))←FLAGS,(IF)←0,(TF)←0 (SP)←(SP)−2,((SP)+1:(SP))←(CS),(CS)←(2) (SP)←(SP)−2,((SP)+1:(SP))←(IP), (IP)←(0) else (AX)←(temp)/(reg16),(DX)←(temp)%(reg16)	U	U	U	U	U	U
		mem16	11110111	mod110 r/m	2-4	(150-168) +EA	(temp)←(DX:AX) if(temp)/(mem16)>FFFFH then (SP)←(SP)−2,((SP)+1:(SP))←FLAGS, IF←0, TF←0 (SP)←(SP)−2,((SP)+1:(SP))←(CS), (CS)←(2) (SP)←(SP)−2,((SP)+1:(SP))←(IP), (IP)←(0) else (AX)←(temp)/(mem16), (DX)←(temp)%(mem16)	U	U	U	U	U	U
	IDIV	reg8	11110110	11111 r/m	2	101-112	(temp)←(AX) if(temp)/(reg8)>0and(temp)/(reg8) >7FH or(temp)/(reg8)>0 and(temp)/(reg8) <0−7FH−1 then (SP)←(SP)−2, (SP)+1:(SP))←FLAGS, (IF)←0, (TF)←0 (SP)←(SP)−2, ((SP)+1:(SP))←(CS), (CS)←(2) (SP)←(SP)−2, ((SP)+1:(SP))←(IP), (IP)←(0) else (AL)←(temp)/(reg8), (AH)←(temp)%(reg8)	U	U	U	U	U	U
		mem8	11110110	mod111 r/m	2-4	(107-118) +EA	(temp)←(AX) if(temp)/(mem8)>0 and(temp)/(mem8) >7FH or(temp)/(mem8)>0 and(temp)/(mem8) <0−7FH−1 then (SP)←(SP)−2,((SP)+1:(SP))←FLAGS,(IF)←0,(TF)←0 (SP)←(SP)−2,((SP)+1:(SP))←(CS), (CS)←(2) (SP)←(SP)−2,((SP)+1:(SP))←(IP), (IP)←(0) else (AL)←(temp)/(mem8), (AH)←(temp)%(mem8)	U	U	U	U	U	U
		reg16	11110111	11111 r/m	2	165-184	(temp)←(DX:AX) if(temp)/(reg16)>0 and(temp)/(reg16) >7FFFH or(temp)/(reg16)<0 and(temp)/(reg16) <0−7FFFH−1 then (SP)←(SP)−2,((SP)+1:(SP))←FLAGS,(IF)←0,(TF)←0 (SP)←(SP)−2,((SP)+1:(SP))←(CS),(CS)←(2) (SP)←(SP)−2,((SP)+1:(SP))←(IP), (IP)←(0) else(AX)←(temp)/(reg16),(DX)←(temp)%(reg16)	U	U	U	U	U	U
		mem16	11110111	mod111 r/m	2-4	(171-190) +EA	(temp)←(DX:AX) if(temp)/(mem16)>0 and(temp)/(mem16) >7FFFH or(temp)/(mem16)<0 and(temp)/(mem16) <0−7FFFH−1 then (SP)←(SP)−2,((SP)+1:(SP))←FLAGS, IF←0, TF←0 (SP)←(SP)−2,((SP)+1:(SP))←(CS), (CS)←(2) (SP)←(SP)−2,((SP)+1:(SP))←(IP), (IP)←(0) else(AX)←(temp)/(mem16),(DX)←(temp)%(mem16)	U	U	U	U	U	U
	AAD		11010101	00001010	2	60	(AL)←(AH)×0AH+(AL) (AH)←0	U	U	U	×	×	×
	CBW		10011000		1	2	if(AL)<80H then(AH)←0 else (AH)←FFH						
	CWD		10011001		1	5	if(AX)<8000H then(DX)←0 else (DX)←FFFFH						
論理演算命令 [illegible]	NOT	reg	1111011W	11010 r/m	2	3	ifW=0 (REG8)←FFH−(REG8) ifW=1 (REG16)←FFFFH−(REG16)						
		mem	1111011W	mod010 r/m	2-4	16+EA	ifW=0 (me8)←FFH−(mem8) ifW=1 (mem16)←FFFFH−(mem16)						
[illegible]	SHL SAL	reg 1	1101000W	11100 r/m	2	2	(CF)←high order dit of(reg), (r/m)←(reg)×2 if high-order dit of (reg)≠(CF) then(OF)←1 else(OF)←0	U	×	×	×	×	×

PC-9801マシン語入門①

命令群	ニーモニック	オペランド	オペレーション・コード 7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	バイト数	クロック数	オペレーション	A	C	O	P	S	Z
論理演算命令 シングル・オペランド論理命令	SHL SAL	mem, 1	1 1 0 1 0 0 0 W	mod 1 0 0 r/m	2 - 4	15 + E A	(CF)←high-order bit of(mem), (mem)←(mem)×2 if high-order bit of (EA) ≠ (CF) then (OF)←1 else (OF)←0	U	×	×	×	×	×
		reg, CL	1 1 0 1 0 0 1 W	1 1 1 0 0 r/m	2	8 + 4/bit	(temp)←(CL), do while (temp) ≠ 0 (CF)←high-order bit of(reg), (reg)←(reg)×2 (temp)←(temp) − 1 (OF) undefined	U	×	U	×	×	×
		mem, CL	1 1 0 1 0 0 1 W	mod 1 0 0 r/m	2 - 4	20 + E A +4/bit	(temp)←(CL), do while (temp) ≠ 0 (CF)←high-order bit of(mem), (mem)←(mem)×2 (temp)←(temp) − 1 (OF) undefined	U	×	U	×	×	×
	SHR	reg, 1	1 1 0 1 0 0 0 W	1 1 1 0 1 r/m	2	2	(CF)←low-order bit of(reg), (reg)←(reg)/2 if high-order bit of(reg) ≠ next-to-high-order bit of(reg) then (OF)←1 else(OF)←0	U	×	×	×	×	×
		mem, 1	1 1 0 1 0 0 0 W	mod 1 0 1 r/m	2 - 4	15 + E A	(CF)←low-order bit of(mem), (mem)←(mem)/2 if high-order bit of(mem) ≠ next-to-high-order bit of(mem) then (OF)←1 else (OF)←0	U	×	×	×	×	×
		reg, CL	1 1 0 1 0 0 1 W	1 1 1 0 1 r/m	2	8 + 4/bit	(temp)←(CL) do while (temp) ≠ 0 (CF)←low-order bit of(reg), (reg)←(reg)/2 (temp)←(temp) − 1 (OF) undefined	U	×	U	×	×	×
		mem, CL	1 1 0 1 0 0 1 W	mod 1 0 1 r/m	2 - 4	20 + E A +4/bit	(temp)←(CL) do while (temp) ≠ 0 (CF)←low-order bit of(mem), (mem)←(mem)/2 (temp)←(temp) − 1 (OF) undefined	U	×	U	×	×	×
	SAR	reg, 1	1 1 0 1 0 0 0 W	1 1 1 1 1 r/m	2	2	(CF)←low-order bit of(reg), (reg)←(reg)/2, (OF)←0	U	×	×	×	×	×
		mem, 1	1 1 0 1 0 0 0 W	mod 1 1 1 r/m	2 - 4	15 + E A	(CF)←low-order bit of(mem), (mem)←(mem)/2, (OF)←0	U	×	×	×	×	×
		reg, CL	1 1 0 1 0 0 1 W	1 1 1 1 1 r/m	2	8 + 4/bit	(temp)←(CL) do while (temp) ≠ 0 (CF)←low-order bit of(reg), (reg)←(reg)/2 (temp)←(temp) − 1 (OF) undefined	U	×	×	×	×	×
		mem, CL	1 1 0 1 0 0 1 W	mod 1 1 1 r/m	2 - 4	20 + E A +4/bit	(temp)→(CL) do while (temp) ≠ 0 (CF)←low-order bit of(mem), (mem)←(mem)/2 (temp)←(temp) − 1 (OF) undefined	U	×	×	×	×	×
	ROL	reg, 1	1 1 0 1 0 0 0 W	1 1 0 0 0 r/m	2	2	(CF)←high-order bit of(reg), (reg)←(reg)×2+(CF) if high-order bit of(reg) ≠ (CF) then (OF)←1 else (OF)←0		×	×			
		mem, 1	1 1 0 1 0 0 0 W	mod 0 0 0 r/m	2 - 4	15 + E A	(CF)←high-order bit of(mem), (mem)←(mem)×2+(CF) if high-order bit of (mem) ≠ (CF) then (OF)←1 else (OF)←0		×	×			
		reg, CL	1 1 0 1 0 0 1 W	1 1 0 0 0 r/m	2	8 + 4/bit	(temp)←(CL) do while (temp) ≠ 0 (CF)←high-order bit of(reg), (reg)×2+(CF) (temp)←(temp) − 1 (OF) undefined		×	U			
		mem, CL	1 1 0 1 0 0 1 W	mod 0 0 0 r/m	2 - 4	20 + E A +4/bit	(temp)←(CF) do while (temp) ≠ 0 (CF)←high order bit of(mem), (mem)←(mem)×2+(CF) (temp)←(temp) − 1 (OF) undefined		×	U			
	ROR	reg, 1	1 1 0 1 0 0 0 W	1 1 0 0 1 r/m	2	2	(CF)←low order bit of(reg), (reg)←(reg)/2 high order bit of (reg)←(CF) if high-order bit of(reg) ≠ next-to-high-order bit of(reg) then (OF)←1 else (OF)←0		×	×			
		mem, 1	1 1 0 1 0 0 0 W	mod 0 0 1 r/m	2 - 4	15 + E A	(CF)←low-order bit of(mem), (mem)←(mem)/2 high order bit of(mem)←(CF) if high-order bit of(mem) ≠ next-to-high-order bit of(mem) then (OF)←1 else (OF)←0		×	×			
		reg, CL	1 1 0 1 0 0 1 W	1 1 0 0 1 r/m	2	8+4/bit	(temp)←(CL) do while (temp) ≠ 0 (CF)←low-order bit of(reg), (reg)←(reg)/2 high order bit of(reg)←(CF) (temp)←(temp) − 1 (OF) undefined		×	U			
		mem, CL	1 1 0 1 0 0 1 W	mod 0 0 1 r/m	2 - 4	20 + E A +4/bit	(temp)←(CL) do while (temp) ≠ 0 (CF)←low-order bit of(mem), (mem)←(mem)/2 high order bit of(mem)←(CF) (temp)←(temp) − 1 (OF) undefined		×	U			
	RCL	reg, 1	1 1 0 1 0 0 0 W	1 1 0 1 0 r/m	2	2	(tmpcf)←(CF), (CF)←high-order bit of(reg) (reg)←(reg)×2+(tmpcf) if high order bit of(reg)←(CF) then (OF)←1 else (OF)←0		×	×			
		mem, 1	1 1 0 1 0 0 0 W	mod 0 1 0 r/m	2 - 4	15 + E A	(tmpcf)←(CF), (CF)←high-order bit of (mem) (mem)←(mem)×2+(tmpcf) if high order bit of(mem)≠(CF) then (OF)←1 else (OF)←0		×	×			
		reg, CL	1 1 0 1 0 0 1 W	1 1 0 1 0 r/m	2	8 + 4/bit	(temp)←(CL) do while (temp) ≠ 0 (tmpcf)←(CF), (CF)←high-order bit of(reg) (reg)←(reg)×2+(tmpcf) (temp)←(temp) − 1 (OF) undefined		×	U			

命令群		ニーモニック	オペランド	オペレーション・コード 76543210	オペレーション・コード 76543210	バイト数	クロック数	オペレーション	A	C	O	P	S	Z
論理演算命令	シングル・オペランド論理命令	RCL	mem, CL	1101001W	mod 010 r/m	2-4	20+EA +4/bit	(temp)←(CL) do while (temp)≠0 (tmpcf)←(CF), (CF)←high-order bit of(mem) (mem)←(mem)※2+(tmpcf) (temp)←(temp)−1 (OF) undefined		×	U			
		RCR	reg, 1	1101000W	11011 r/m	2	2	(tmpcf)←(CF), (CF)←low-order bit of(reg), (reg)←(reg)/2 high-order bit of (reg)←(tmpcf) if high-order bit of(reg) ≠ next-to-high-order bit of(reg) then (OF)←1 else (OF)←0		×	×			
			mem, 1	1101000W	mod 011 r/m	2-4	15+EA	(tmpcf)←(CF), (CF)←low-order bit of(mem), (mem)←(mem)/2 high-order bit of(mem)←(tmpcf) if high-order bit of(mem) ≠ next-to-high-order bit of(mem) then (OF)←1 else (OF)←0		×	×			
			reg, CL	1101001W	11011 r/m	2	8+ +4/bit	(temp)←(CL) do while (temp)≠0 (tmpcf)←(CF), (CF)←low-order bit of(reg) (reg)←(reg)/2 high-order bit of(r/m)←(tmpcf) (temp)←(temp)−1 (OF) undefined		×	U			
			mem, CL	1101001W	mod 011 r/m	2-4	20+EA +4/bit	(temp)←(CL) do while (temp)≠0 (tmpcf)←(CF), (CF)←low-order bit of(mem) (mem)←(mem)/2 high-order bit of(mem)←(tmpcf) (temp)←(temp)−1 (OF) undefined		×	U			
	ダブル・オペランド論理命令	AND	reg, reg	0010000W	11 reg r/m	2	3	(reg)←(reg)&(reg)	U	0	0	×	×	×
			mem, reg	0010000W	mod reg r/m	2-4	16+EA	(mem)←(mem)&(reg)	U	0	0	×	×	×
			reg, mem	0010001W	mod reg r/m	2-4	9+EA	(reg)←(reg)&(mem)	U	0	0	×	×	×
			reg, imm	1000000W	11100 r/m	3-4	4	(reg)←(reg)& data	U	0	0	×	×	×
			mem, imm	1000000W	mod 100 r/m	3-6	17+EA	(mem)←(mem)& data	U	0	0	×	×	×
			acc, imm	0010010W		2-3	4	ifW=0 (AL)←(AL)& data ifW=1 (AX)←(AX)& data	U	0	0	×	×	×
		TEST	reg, reg	1000010W	11 reg r/m	2	3	(reg)&(reg)	U	0	0	×	×	×
			mem, reg	1000010W	mod reg r/m	2-4	9+EA	(reg)&(mem)	U	0	0	×	×	×
			reg, imm	1111011W	11000 r/m	3-4	5	(reg)& data	U	0	0	×	×	×
			mem, imm	1111011W	mod 000 r/m	3-6	11+EA	(mem)& data	U	0	0	×	×	×
			acc, imm	1010100W		2-3	4	ifW=0 (AL)& data ifW=1 (AX)& data	U	0	0	×	×	×
		OR	reg, reg	0000100W	11 reg r/m	2	3	(reg)←(reg)∨(reg)	U	0	0	×	×	×
			mem, reg	0000100W	mod reg r/m	2-4	16+EA	(mem)←(mem)∨(reg)	U	0	0	×	×	×
			reg, mem	0000101W	mod reg r/m	2-4	9+EA	(reg)←(reg)∨(mem)	U	0	0	×	×	×
			reg, imm	1000000W	11001 r/m	3-4	4	(reg)←(reg)∨ data	U	0	0	×	×	×
			mem, imm	1000000W	mod 001 r/m	3-6	17+EA	(mem)←(mem)∨ data	U	0	0	×	×	×
			acc, imm	0000110W		2-3	4	ifW=0 (AL)←(AL)∨ data ifW=1 (AX)←(AX)∨ data	U	0	0	×	×	×
		XOR	reg, reg	0011000W	11 reg r/m	2	3	(reg)←(reg)∀(reg)	U	0	0	×	×	×
			mem, reg	0011000W	mod reg r/m	2-4	16+EA	(mem)←(mem)∀(reg)	U	0	0	×	×	×
			reg, mem	0011001W	mod reg r/m	2-4	9+EA	(reg)←(reg)∀(mem)	U	0	0	×	×	×
			reg, imm	1000000W	11110 r/m	3-4	4	(reg)←(reg)∀ data	U	0	0	×	×	×
			mem, imm	1000000W	mod 110 r/m	3-6	17+EA	(mem)←(mem)∀ data	U	0	0	×	×	×
			acc, imm	0011010W		2-3	4	ifW=0 (AL)←(AL)∀ data ifW=1 (AX)←(AX)∀ data	U	0	0	×	×	×
ストリング命令	ストリング・プリフィックス命令	REP REPE REPZ		11110011		1	2	do while (CX)≠0 続くバイトのプリミティブ・ストリング命令を実行する (CX)←(CX)−1 保留割込みがあれば処理する プリミティブ命令がCMPSまたはSCASでかつ, (ZF)≠1のときループを抜け出る						
		REPNE REPNZ		11110010		1	2	do while (CX)≠0 続くバイトのプリミティブ・ストリング命令を実行する (CX)←(CX)−1 保留割込みがあれば処理する (ZF)≠0 のときループを抜け出る						
	プリミティブ・ストリング命令	MOVS MOVSB* MOVSW*	dst-string, src-string	1010010W		1	9+ 17/rep /18	ifW=0 ((DI))←((SI)) if(DF)=0 then (SI)←(SI)+1, (DI)←(DI)+1 else(SI)←(SI)−1, (DI)←(DI)−1 ifW=1 ((DI)+1 (DI))←((SI)+1 (SI)) if(DF)=0 then(SI)←(SI)+2, (DI)←(DI)+2 else(SI)←(SI)−2, (DI)←(DI)−2						

命令群		ニーモニック	オペランド	オペレーション・コード		バイト数	クロック数	オペレーション	フラグ					
				76543210	76543210				A	C	O	P	S	Z
ストリング命令	プリミティブ・ストリング命令	CMPS	dst-string, src-string	1010011W		1	9+22/rep/22	ifW=0 ((SI))−((DI)) if(DF)=0 then(SI)←(SI)+1, (DI)←(DI)+1 else(SI)←(SI)−1, (DI)←(DI)−1 ifW=1 ((SI)+1 : (I))−((DI)+1 : (DI)) if(DF)=0 then(SI)←(SI)+2, (DI)←(DI)+2 else(SI)←(SI)−2, (DI)←(DI)−2	×	×	×	×	×	×
		SCAS	dst-string	1010111W		1	9+15/rep/15	ifW=0 (AL)−((DI)) if(DF)=0 then(DI)←(DI)+1 else(DI)←(DI)−1 ifW=1 (AX)−((DI)+1 : (DI)) if(DF)=0 then(DI)←(DI)+2 else(DI)←(DI)−2	×	×	×	×	×	×
		LODS	src-string	1010110W		1	9+13/rep/12	ifW=0 (AL)←((SI)) if(DF)=0 then(SI)←(SI)+1 else(SI)←(SI)−1 ifW=1 (AX)←((SI)+1 : (SI)) if(DF)=0 then(SI)←(SI)+2 else(SI)←(SI)−2						
		STOS	dst-string	1010101W		1	9+10/rep/11	ifW=0 ((DI))←(AL) if(DF)=0 then(DI)←(DI)+1 else(DI)←(DI)−1 ifW=1 ((DI)+1 : (DI))←(AX) if(DF)=0 then(DI)←(DI)+2 else(DI)←(DI)−2						
分岐命令	制御転送命令	CALL	near-proc	11101000		3	19	(SP)←(SP)−2 ((SP)+1 : (SP))←(IP), (IP)←(IP)+disp						
			regptr16	11111111	11010 r/m	2	16	(SP)←(SP)−2 ((SP)+1 : (SP))←(IP), (IP)←(regtr16)						
			memptr16	11111111	mod010 r/m	2-4	21+EA	(SP)←(SP)−2 ((SP)+1 : (SP))←(IP), (IP)←memptr16						
			far-proc	10011010		5	28	(SP)←(SP)−2 ((SP)+1 : (SP))←(CS), (CS)←seg (SP)←(SP)−2 ((SP)+1 : (SP))←(IP), (IP)←offset						
			memptr32	11111111	mod011 r/m	2-4	37+EA	(SP)←(SP)−2 ((SP)+1 : (SP))←(CS), (CS)←(mmptr32+2) (SP)←(SP)−2 ((SP)+1 : (SP))←(IP), (IP)←(memptr32)						
		JMP	near-label	11101001		3	15	(IP)←(IP)+disp						
			short-label	11101011		2	15	(IP)←(IP)+disp						
			regptr16	11111111	11100 r/m	2	11	(IP)←(regptr16)						
			memptr16	11111111	mod100 r/m	2-4	18+EA	(IP)←(memptr16)						
			far-label	11101010		5	15	(CS)←seg (IP)←offset						
			memptr32	11111111	mod101 r/m	2-4	24+EA	(CS)←(memptr+2) (IP)←(memptr32)						
		RET		11000011		1	8	(IP)←((SP)+1 : (SP)) (SP)←(SP)+2						
			pop-value	11000010		3	12	(IP)←((SP)+1 : (SP)) (SP)←(SP)+2, (SP)←(SP)+data						
				11001011		1	18	(IP)←((SP)+1 : (SP)) (SP)←(SP)+2 (CS)←((SP)+1 : (SP)) (SP)←(SP)+2						
			pop-value	11001010		3	17	(IP)←((SP)+1 : (SP)) (SP)←(SP)+2 (CS)←((SP)+1 : (SP)) (SP)←(SP)+2, (SP)←(SP)+data						
	条件付制御転送命令	JE JZ	short-label	01110100		2	16/4	if(ZF)=1 (IP)←(IP)+disp						
		JL JNGE	short-label	01111100		2	16/4	if(SF)⊻(OF)=1 (IP)←(IP)+disp						
		JLE JNG	short-label	01111110		2	16/4	if [(SF)⊻(OF)]∨(ZF)=1 (IP)←(IP)+disp						
		JB JNAE	short-label	01110010		2	16/4	if(CF)=1 (IP)←(IP)+disp						
		JBE JNA	short-label	01110110		2	16/4	if(CF)∨(ZF)=1 (IP)←(IP)+disp						
		JP JPE	short-label	01111010		2	16/4	if(PF)=1 (IP)←(IP)+disp						

命令群		ニーモニック	オペランド	オペレーション・コード 76543210	76543210	バイト数	クロック数	オペレーション	フラグ A	C	O	P	S	Z
分岐命令	条件付制御転送命令	JO	Short-label	01110000		2	16/4	if(OF)=1 (IP)←(IP)+disp						
		JS	Short-label	01111000		2	16/4	if(SF)=1 (IP)←(IP)+disp						
		JNE JNZ	Short-label	01110101		2	16/4	if(ZF)=0 (IP)←(IP)+disp						
		JNL JGE	Short-label	01111101		2	16/4	if(SF)⊻(OF)=0 (IP)←(IP)+disp						
		JNLE JG	Short-label	01111111		2	16/4	if ((SF)⊻(OF))∨(ZF)=0 (IP)←(IP)+disp						
		JNB JAE	Short-label	01110011		2	16/4	if(CF)=0 (IP)←(IP)+disp						
		JNBE JA	Short-label	01110111		2	16/4	if(CF)∨(ZF)=0 (IP)←(IP)+disp						
		JNP JPO	Short-label	01111011		2	16/4	if(PF)=0 (IP)←(IP)+disp						
		JNO	Short-label	01110001		2	16/4	if(OF)=0 (IP)←(IP)+disp						
		JNS	Short-label	01111001		2	16/4	if(SF)=0 (IP)←(IP)+disp						
	くり返し転送命令	LOOP	Short-label	11100010		2	17/5	(CX)←(CX)−1 if(CX)≠0 (IP)←(IP)+disp						
		LOOPZ LOOPE	Short-label	11100001		2	18/6	(CX)←(CX)−1 if(ZF)=1and(CX)≠0 (IP)←(IP)+disp						
		LOOPNZ LOOPNE	Short-label	11100000		2	19/5	(CX)←(CX)−1 if(ZF)=0 and(CX)≠0 (IP)←(IP)+disp						
		JCXZ	Short-label	11100011		2	18/6	if(CX)=0 (IP)←(IP)+disp						
	割込転送命令	INT	3	11001100		1	52	(SP)←(SP)−2 ((SP)+1:(SP))←FLAGS, (IF)←0, (TF)←0 (SP)←(SP)−2 ((SP)+1:(SP))←(CS), (CS)←(14) (SP)←(SP)−2 ((SP)+1:(SP))←(IP), (IP)←(12)						
			imm8 (*3)	11001101		2	51	(SP)←(SP)−2 ((SP)+1:(SP))←FLAGS, (IF)←0, (TF)←0 (SP)←(SP)−2 ((SP)+1:(SP))←(CS), (CS)←(type×4+2) (SP)←(SP)−2 ((SP)+1:(SP))←(IP), (IP)←(type×4)						
		INTO		11001110		1	53/4	if(OF)=1 (SP)←(SP)−2 ((SP)+1:(SP))←FLAGS, (IF)←0, (TF)←0 (SP)←(SP)−2 ((SP)+1:(SP))←(CS), (CS)←(12H) (SP)←(SP)−2 ((SP)+1:(SP))←(IP), (IP)←(10H)						
		IRET		11001111		1	24	(IP)←((SP)+1:(SP)), (SP)←(SP)+2 (CS)←((SP)+1:(SP)), (SP)←(SP)+2 FLAGS←((SP)+1:(SP)), (SP)←(SP)+2	×	×	×	×	×	×
CPU/フラグ制御命令	フラグ制御命令	CLC		11111000		1	2	(CF)←0		0				
		CMC		11110101		1	2	(CF)←$\overline{(CF)}$		×				
		STC		11111001		1	2	(CF)←1		1				
		CLD		11111100		1	2	(DF)←0						
		STD		11111101		1	2	(DF)←1						
		CLI		11111010		1	2	(IF)←0						
		STI		11111011		1	2	(IF)←1						
	CPU制御命令	HLT		11110100		1	2	CPU halt						
		WAIT		10011011		1	3+5n	CPU wait						
		ESC	ext-op, reg	11011XXX	11YYY r/m	2	2	CPU escape data bus←(reg)						
			ext-op, mem	11011XXX	modYYY r/m	2-4	8+EA	CPU escape data bus←(mem)						
		LOCK		11110000		1	2	Bus Lock Prefix						
		NOP		10010000		1	3	ノー・オペレーション						

ASCII SYSTEMS BANK

内容抜粋

8086の構成
マシン語プログラミングの予備知識
8086の動作を見る
DEBUGと機械語モニタの相違点
アルゴリズムとフローチャート
データの転送とアドレッシングモード
データの比較と分岐
データの演算とフラグ
データの検索とソーティング
マシン語によるV-RAMのアクセス
MASMの概要
MASMの用法
EDLINの使用法
アセンブルの実際
BASICとマシン語のリンク
BASICとメモリマップ
マシン語サブルーチンコールと引数
BASIC-ROMの利用
ハードウェア構成と割込み
ハードウェア割込み
グラフィック・パターンの移動
行列演算ルーチン
RAMディスク
簡単なインタープリタをつくる
MS-DOS・BASICトランスファープログラム
MASMの高度な利用法

岩瀬正幸・藤井敬雄共著

アスキー・システム・バンク(PC98#2)

PC-9801 マシン語入門

定価2,580円(本体2,505円)
ISBN4-87148-775-X
C3055 P2580E